UNIVERSIDAD POLITÉCNICA DE MADRID

Escuela Universitaria de Ingeniería Técnica de Telecomunicación

TRABAJO FIN DE MÁSTER

Máster en Ingeniería de Sistemas y Servicios para la Sociedad de la Información

Energy- and Power-aware Scheduling Algorithm Modeling for Battery-powered Embedded Systems

Jianguo Wei

julio de 2012

To learn without thinking, one will be lost in his learning.

To think without learning, one will be imperilled.

Confucius

Acknowledgements

First of all, I would like to thank all the professors and lab mates in GDEM. My first thank to my advisor, Eduardo Juárez Martínez, for his advancing on my research work, his patient guidance of my development and growth, and his encouragement on my study as well as everyday life. He provides me valuable and quick comments for improving my research work. He reviews the state of my research work every week and we have spent many time discussing and arguing some points of the work. Thanks also to the other professors in my research group: César Sanz Álvaro, Matías Javier Garrido González, Fernando Pescador del Oso and Pedro José Lobo Perea, for their efforts in providing and creating such a friendly and helpful working environment in the lab. Thanks to Rong Ren, my only Chinese lab mate in the lab. I have had a nice experience to work together with her and benefit a lot by discussing with her. Thanks to my former lab mates David, Gonzalo and Ernesto, for their generous help both in the work and in everyday life. Without their help, my life in Madrid would be much tougher and boring. Thanks to my lab mates Juanjo, Enrique, Miguel and Arnaud, I´ve had a fantastic time together with them during the past year. I enjoy talking with you during the lunch and appreciate your efforts to push me speak more Spanish. Thanks to Oscar, a trustworthy lab mate who is always ready to help me out.

Secondly, I would like to thank my family and friends. Thanks to my parents, for their hard working to bring me up and ensure my receiving a good-quality of education, for their greatness and unselfishness to let their only child fly as far as he can and freely chase his life and future. Thanks to my best Chinese friends in Madrid, Rong Ren, Shan Huang, Meijuan Zhang, Liang Chai and Xiang Wang, for the pure friendship. Remember the first night we spent together in Beijing airport, and the uncountable nights we spent together with or without a reason. Thanks to my great chef and best friend, Qingxing Cai, for calling me every day for dinner, whenever late I go back home, I know there are delicious dishes waiting for me. Thanks to my flat mates Qi Zhao and Zhaoxin Sun, it is a pleasure to live and enjoy exciting football matches together with you.

Table of Contents

List of Figures

List of Tables

Resumen

La gestión de energía en los sistema móviles está considerada hoy en día como un reto fundamental, notándose, especialmente, en aquellos terminales que utilizando un sistema operativo implementan múltiples funciones. Es común en los sistemas móviles actuales ejecutar simultaneamente diferentes aplicaciones y tener, para una de ellas, un objetivo de tiempo de uso de la batería. Tradicionalmente, las políticas de gestión de consumo de potencia de los sistemas operativos hacen lo que está en sus manos para ahorrar energía y satisfacer sus requisitos de prestaciones, pero no son capaces de proporcionar un objetivo de tiempo de utilización del sistema, dejando al usuario la difícil tarea de buscar un compromiso entre prestaciones y tiempo de utilización del sistema. Esta tesis, como contribución, proporciona una nueva manera de afrontar el problema. En ella se establece que un esquema de gestión de consumo de energía debería, en primer lugar, garantizar, para una aplicación dada, un tiempo mínimo de utilización de la batería que estuviera especificado por el usuario, restringiendo la potencia media consumida por las aplicaciones que se puedan considerar menos importantes y, en segundo lugar, maximizar las prestaciones globales sin comprometer la garantía de utilización de la batería. Como soporte de lo anterior, la energía, en lugar del tiempo de CPU o el ancho de banda, debería gestionarse globalmente por el sistema operativo como recurso de primera clase.

Como primera fase en el desarrollo completo de un esquema de gestión de consumo, esta tesis presenta un algoritmo de planificación de encolado equitativo (*fair queueing*) basado en el consumo de energía, es decir, una nueva clase de algoritmos de planificación que, en combinación con mecanismos que restrinjan la tasa de descarga de una batería, gestionen de forma sistemática la energía como recurso de primera clase, con el objetivo de garantizar, para una aplicación dada, un tiempo de uso de la batería, definido por el usuario, en sistemas móviles empotrados. El encolado equitativo de energía es una extensión al dominio de la energía del encolado equitativo tradicional. Esta clase de algoritmos asigna una reserva de potencia a cada tarea y gestiona la energía sirviéndola de manera proporcional a su reserva. Este uso proporcional de la energía garantiza que cada tarea reciba una porción de potencia y evita que haya tareas que se vean privadas de recibir energía por otras con un comportamiento más ambicioso. Esta clase de algoritmos trata a todas las tareas por igual y puede planificar tareas periódicas en tiempo real asignando a cada una de ellas una reserva de potencia que es adecuada para proporcionar la mayor de las

cantidades de energía demandadas por período. Sin embargo, es posible demostrar que sólo se consigue cumplir con los requisitos impuestos por todos los plazos temporales con reservas de potencia extremadamente conservadoras. En esta tesis, para proporcionar un soporte más flexible y eficiente para diferentes tipos de tareas de tiempo real junto con el resto de tareas, se combina un mecanismo de planificación basado en prioridades con el encolado equitativo basado en energía. En esta clase de algoritmos, gracias al método introducido, que controla el tiempo que se ejecuta con prioridad una tarea de tiempo real, se puede establecer un compromiso entre el cumplimiento de los requisitos de tiempo real y el consumo de potencia.

Para evaluar los algoritmos, se ha diseñado en SystemC un banco de pruebas. Los resultados muestran que el algoritmo de encolado equitativo basado en el consumo de energía consigue el balance entre el uso proporcional a la energía reservada y el cumplimiento de los requisitos de tiempo real.

Summary

Energy management has always been recognized as a challenge in mobile systems, especially in modern OS-based mobile systems where multi-functioning are widely supported. Nowadays, it is common for a mobile system user to run multiple applications simultaneously while having a target battery lifetime in mind for a specific application. Traditional OS-level power management (PM) policies make their best effort to save energy under performance constraint, but fail to guarantee a target lifetime, leaving the painful trading off between the total performance of applications and the target lifetime to the user itself. This thesis provides a new way to deal with the problem. It is advocated that a strong energy-aware PM scheme should first guarantee a user-specified battery lifetime to a target application by restricting the average power of those less important applications, and in addition to that, maximize the total performance of applications without harming the lifetime guarantee. As a support, energy, instead of CPU or transmission bandwidth, should be globally managed as the first-class resource by the OS.

As the first-stage work of a complete PM scheme, this thesis presents the energy-based fair queuing scheduling, a novel class of energy-aware scheduling algorithms which, in combination with a mechanism of battery discharge rate restricting, systematically manage energy as the first-class resource with the objective of guaranteeing a user-specified battery lifetime for a target application in OS-based mobile systems. Energy-based fair queuing is a cross-application of the traditional fair queuing in the energy management domain. It assigns a power share to each task, and manages energy by proportionally serving energy to tasks according to their assigned power shares. The proportional energy use establishes proportional share of the system power among tasks, which guarantees a minimum power for each task and thus, avoids energy starvation on any task. Energy-based fair queuing treats all tasks equally as one type and supports periodical time-sensitive tasks by allocating each of them a share of system power that is adequate to meet the highest energy demand in all periods. However, an overly conservative power share is usually required to guarantee the meeting of all time constraints. To provide more effective and flexible support for various types of time-sensitive tasks in general purpose operating systems, an extra real-time friendly mechanism is introduced to combine priority-based scheduling into the energy-based fair queuing. Since a method is available to control

the maximum time one time-sensitive task can run with priority, the power control and time-constraint meeting can be flexibly traded off.

A SystemC-based test-bench is designed to assess the algorithms. Simulation results show the success of the energy-based fair queuing in achieving proportional energy use, time-constraint meeting, and a proper trading off between them.

1 Introduction

In this chapter, we introduce the thesis work from a general overview. We start in section 1.1 with an introduction of the research problem, and then continue with section 1.2 to discuss our motivation to solve the problem. Section 1.3 gives a brief summary of the contribution of this thesis work. Finally section 1.4 provides the organization of the whole thesis.

1.1 Powerful but no lifetime guarantee

The energy demand of applications is increasingly higher in modern mobile systems due to the growth in CPU frequency, transmission bandwidth and software complexity; however, battery capacity is not experiencing a significant augment. On the contrary, the relentless trend to make mobile devices lighter and smaller further restricts the battery capacity. As a consequence, energy in modern mobile systems has become a limited resource as important as CPU, memory and network bandwidth [1], [2] and [3].

To resolve the energy issue, power management (PM) schemes from the operating system and software level have been widely researched. The main work of a PM scheme is to trade off between the application performance and energy consumption. Depending on which factor is more concerned, PM schemes can have different levels of energy awareness, that is, the extent to which a PM scheme is able to be aware of the system power state. A performance-centric PM scheme will first allow applications to consume energy as demanded to guarantee their minimum performance, and once this goal is achieved energy saving is considered. In this case, the CPU or transmission bandwidth is managed as the first-class resource all around the system. Power optimization is put on a secondary position and, then, only best effort strategies are applied without guaranteeing a specific target lifetime, which leads to a weak awareness of the system energy state. Most PM schemes in modern mainstream operating systems are performance-centric, and generally fall into two categories: dynamic power management (DPM) [4], [5] and [6] that runs the workload to completion at the maximum CPU speed and then rests the system in the longest low-power mode; and dynamic voltage and frequency scaling (DVFS) [7], [8], [9] and [10] that assumes the highest energy saving can be achieved by running at the lowest performance setting under deadline constraints.

One notable feature of modern OS-based mobile systems is the multi-functionality. Thanks to the significant improvement on hardware, it is now very common for a user to have multiple applications running simultaneously on a powerful Smartphone or tablet. For example, we may be browsing the websites while having the push notifications on to keep informed the latest states from the email and facebook. However, each single application is consuming energy. If the system has all the applications continuously running without limiting their powers, the battery will soon be depleted. Not only does the usefulness of a mobile system depend on the CPU speed,

application functionality and quality, but also is limited by its lifetime. A mobile system user often has a reasonable estimate of how long the battery needs to last for a specific application, and there are many scenarios in which a specific battery lifetime is a more important factor than the total application performance. Imagine a businessperson is adjusting the slides on his way to the product presentation while having the music on for inspiration and relaxation, lifetime of the laptop is more concerned than the quality of the music and brightness of the screen; or a football fan is watching a live football match broadcast while having a forum webpage on for maintaining an interaction with other fans, a lifetime of the mobile device that lasts the length of the football match is more concerned than the video quantity and webpage update rate. Traditional performance-centric PM schemes make their best effort to save energy but leave the painful trading off between user experience and target lifetime to the user itself. To further save energy and extend the battery lifetime, the user has to manually turn off some currently unused or less important functions like the wifi, GPS or data push notifications. However, without the information and control from the OS side, a user can easily fall into a dilemma: on one side, load a new application or run existing applications in higher quantity may undermine the targeted lifetime; on the other side, keep the original system setting may lose the chance to enhance user experience within the target lifetime.

In summary, since the CPU or transmission bandwidth instead of the energy is managed as the first-class resource, PM schemes in modern OS-based mobile systems are of weak energy awareness considering that they first fail to provide a lifetime guarantee, and then fail to provide an automatic tradeoff between the target lifetime and the application performance.

1.2 Motivation

We believe the operating system, instead of the user, should take the responsibility to guarantee a target lifetime to the mobile system. Moreover, considering the power-related system states and application properties are all obtainable by the operating system, PM schemes from the OS level can be properly designed to achieve the optimal tradeoff between the target lifetime and the application performance.

The objective of our research is to develop OS-based PM schemes with strong energy awareness. A strong energy-aware PM scheme should be able to first guarantee a user-specified battery lifetime for the most important application by

restricting the average powers (and implicitly the energy consumption) of those less important applications if necessary, and in addition to that, achieve more advanced energy-related goals regarding application performance maximization. Necessarily, power optimization should be put on the most important position, and energy, instead of CPU or transmission bandwidth, should be considered as the first-class resource [1] and [3]. While CPU and transmission bandwidth are exclusive to one device, energy is global to all devices and has an impact to every resource in the system. Managing energy globally as the first-class resource brings more opportunities to power optimization and thus, extends the design space for developing strong energy-aware PM schemes.

To guarantee a user-specified battery lifetime for a target application, a PM scheme should be able to properly distribute the energy both along the time and among the applications. A proper energy distribution along the time establishes the average system power required to achieve a target lifetime; while a proper energy distribution among different applications at least guarantees to the target application an energy allocation that is consistent to its actual energy demand. Specifically, when the energy available in the battery is adequate to achieve the target lifetime while meet the energy demands of all applications, each application should be allocated an share of energy that is appropriately proportional and consistent with its actual energy demand; otherwise, the energy allocation, and implicitly the average power, of those less important tasks should be restricted to guarantee an adequate energy allocation for the target application. A target application that is self-adaptive can choose to degrade its quantity and correspondingly reduce its energy demand to leave more energy allocation to the other applications.

Applications can be allocated shares of energy that are proportional and consistent to their energy demands, however, their ability to consume energy proportionally depends on the schedulers that control access to the devices. To avoid any application monopolizing the devices and spending its energy allocation much faster than others, the scheduling policy should achieve energy use proportional to the allocated energy share of each task. Proportional energy consumption indicates proportional system power sharing among applications, which guarantees a minimum power for each application and thus, avoids energy starvation on any application when there is adequate energy available for meeting the energy demand of all tasks under the target lifetime constraint.

In a general purpose operating system, maximize the total application performance concerns the different performance measurement methods of various applications. Generally we consider three types of applications: they are periodical real-time, interactive and regular applications. First, for all types of applications, the performance partially depends on the level of quality one application provides. A higher quality of service indicates the requirement of a larger energy share. Then, the performance of each application is further measured according to its own property. For a real-time application, it is measured by the number of deadlines met; for an interactive application, it is measured by the response time of each request; and for a regular application, it is measured by the total time required for completing the work. It is a difficult job whether to quantify and measure the total performance of all applications, or to maximize it by balancing the performance of different applications. However, under the target lifetime constraint, we can still make some efforts to improve the total application performance without incurring the above inconvenience. First, minimize the residual energy left in the battery when the target lifetime is achieved. Minimum residual energy indicates maximum energy available for allocation during the target lifetime, thus, supports applications of higher quality or larger number of applications; while too much residual energy indicates an overly conservative resource management and lost opportunities for improved performance [3]. Second, maximize the real-time performance of real-time and interactive applications with a given share of energy and system power. With a given share of energy under proportional power sharing, the performance of a regular task is constant because the total time for completing its work only depends on the given share of system power; while the real-time performance of a real-time or interactive application is variable depending on how timely the energy is received. A real-time application receives a higher energy share after its deadlines may have worse performance than receiving a lower energy share but with all the deadlines met. Therefore, besides of achieving proportional system power sharing, the scheduling policies should also be designed to meet time constraints under general purpose operating systems.

Proportional power sharing guarantees a minimum power for each application and at the same time restricts the power of each application to avoid any of them excessively consuming energy within certain period of time. However, a strict power restriction can be in conflict with the time-constraint meeting of a soft real-time application (e.g. multimedia application), in which the energy demand, and correspondingly the required power, of each period can be significantly different. On one hand, guaranteeing the task an overly large power that meets its high energy

demand (and correspondingly meets the deadlines) brings the risk to energy starve other tasks; on the other hand, a power guarantee that fails to meet the energy demand of most periods also fails to meet most of the deadlines. Therefore, certain mechanisms should also be available to provide a flexible trade-off between the power control and time-constraint meeting for soft real-time tasks.

1.3 Contributions

This thesis work extends the traditional fair queuing algorithms to the energy-management domain, and presents the energy-based fair queuing, a novel class of energy-aware scheduling algorithms that achieve proportional power sharing among tasks and provide real-time performance support in general purpose operating systems. Energy-based fair queuing manages energy as the first-class resource globally in the system and allocates energy to tasks proportional to their assigned power shares; by combining it with an energy allocation mechanism that restricts the discharge rate of the battery, a strong energy-aware PM scheme is proposed to achieve a user-specified battery lifetime for a target application. Besides, a real-time friendly mechanism is combined into the energy-based fair queuing to provide more effective and flexible support for various types of time-sensitive applications. Additionally, the real-time friendly mechanism provides a flexible method to trade off the energy consumption management and time-constraint meeting by properly adjusting certain parameter.

1.4 Organization

The remaining of this thesis is structured as follows. Chapter 2 introduces the related literatures. Chapter 3 first briefly introduces a mechanism that throttles the energy dissipation to guarantee a target lifetime, then presents the energy-based fair queuing scheduling and later discusses in detail its power management and time-constraint meeting, finally introduces a real-time friendly mechanism to improve its support for time-sensitive tasks. Chapter 4 describes the design of the test-bench for simulation. The setting up of simulation experiments and analysis of simulation results are given in chapter 5. Finally, chapter 6 concludes this thesis and discusses directions for future research.

2 Related Work

In this chapter, we review selected research literatures related to the topics covered in this thesis. We start in section 2.1 with a summary of several studies about power management schemes that aim to guarantee a target lifetime for OS-based mobile systems. We continue with section 2.2 with an overview on real-time scheduling algorithms that are commonly employed in general purpose operating systems to support time-sensitive tasks. Section 2.3 provides a basis on fair queuing scheduling and a comparison on different fair queuing algorithms applied in network and CPU scheduling.

2.1. Lifetime-oriented power management

Lifetime-oriented power management (PM) schemes aim to guarantee a target lifetime for mobile systems. To achieve that, the energy is usually managed as the first-class resource or at least raised to a position of equivalent importance as other system resources such as CPU and network bandwidth. Different from energy-efficient schemes such as dynamic voltage and frequency scaling (DVFS) and dynamic power management (DMP) that first guarantees user-acceptable Quality of Service (QoS) and then save energy as much as possible, lifetime-oriented PM schemes set their primary goal as guaranteeing a target battery lifetime and once this goal is achieved, improve the QoS with their best efforts. Generally, there are two types of lifetime-oriented PM schemes: energy-centric scheduling schemes [1], [3] and [12] that raise the target lifetime to the first-class status among performance goals, and schemes that treat an energy goal as important as other performance goals [2] and [11]. The later ones usually rely on the cooperation of applications that are modified to be energy-aware [12].

Ellis and Vahdat [13] and [14] were among the first ones that realize energy should be explicitly managed by the operating system as a first-class resource to develop powerful PM schemes. They proposed that applications should be involved into the OS-level power management, and in supporting that, OS/application interfaces [15] as well as energy accounting tools [16] should be developed. Flinn and Satyanarayanan [11] first developed and implemented on the Odyssey platform [17] a PM scheme based on application adaptation to guarantee a battery lifetime. To achieve the energy goal, Odyssey periodically measures the residual energy available in the battery, predicts future energy demand based on present and past power usage, and notifies applications with an upcall if adaptation of energy demand is needed. A similar PM scheme was later developed by Neugebauer and McAuley [2] on Nemesis. Nemesis also requires applications to be energy-aware and cooperative, but introduces an economic model to provide feedback to applications. Odyssey and Nemesis provides a method from the application side to achieve a target battery lifetime, but the space to set a user-desired duration is limited by the number of applications that support adaptation as well as the minimum level of degradation that the user can accept. Besides, the requirement of applications to be adaptive and energy-aware impedes this method to be widely applied in general systems.

Zeng and Ellis introduce in [3] an energy-centric scheduling algorithm adapted from stride scheduling [18], that, combined with their model of energy abstraction called Currentcy, achieves a target lifetime for mobile devices. In the Currentcy model, energy is systematically managed as a first-class resource without requiring the application to be energy-aware, and the idea of limiting energy within each epoch was first proposed to guarantee a target operational time. Two modules, namely the allocation module and the scheduling module work cooperatively to share the limited Currentcy among competing task. While the allocation module allocates the total Currentcy available in each epoch to different tasks according to user-specified proportion, the scheduling module provides opportunities for the tasks to fairly spend the allocated Currentcy. Each task has a container that adapts the capacity based on its historical energy consumption. The container saves the unspent Currentcy from last epoch and receives Currentcy at the beginning of a new epoch until the capacity is reached. Then, within each epoch, tasks can proportionally consume energy according to their shares until exhausting the Currentcy in their containers. Although the Currentcy model provides a solution from the operating system side to guarantee a target operational time, the applications considered in the model are general ones with little information about the task workload such as deadlines. This makes the scheduling algorithm inapplicable to multimedia applications, one of our main reference applications. Besides, for tasks that need to be continuously active, it is not appreciated to enforce them into idle once their Currentcy are exhausted. Although there is a mechanism of adaptive capacity available to reallocate energy based on the energy consumption history of different tasks, it mainly aims to reduce the residual energy, therefore, is not strong enough in guaranteeing an adequate energy allocation to keep a task active during the whole epoch. Zeng and Ellis in paper [12] suggested that the Currentcy model can be combined with application adaption approaches and energy-efficient algorithms such as DVFS to form more sophisticated PM schemes.

2.2. Real-time scheduling

2.2.1 Rate-monotonic (RM) and Earliest deadline first (EDF)

Real-time scheduling algorithms such as rate-monotonic (RM) scheduling [19] and [20] and earliest deadline first (EDF) scheduling [20] and [21] are designed for better supporting real-time tasks and meeting their timeliness requirements. Rate-monotonic (RM) scheduler and its static priority counterparts are simple to implement

and efficient in providing real-time guarantees provided that the periods and workloads of tasks are known in advance and the operating system is preemptive [22]. In particular, earliest deadline first (EDF) is optimal when system is under-loaded because theoretically it is believed that if any scheduling algorithms can meet all the deadlines then EDF can [23]. When the system is overloaded, with EDF the set of tasks that will miss their deadlines is largely unpredictable, but with fixed priority scheduler low-priority processes tend to miss their deadlines while high-priority processes still meet their deadlines. Those approaches can be found widely applied in real-time embedded system. However, they cannot be applied to conventional tasks scheduling because real-time constraints are required for determining the execution order. One tempting solution for this problem is adding periodic deadlines to conventional tasks, but artificial constraints are unnecessarily introduced thus reducing the effectiveness of the system [24].

In order to support multimedia applications, general-purpose operating systems (e.g. Linux) separate tasks into different classes, and real-time tasks are set to have strictly higher static priority than any other class of tasks so that they are able to obtain processor resources when needed in order to meet their time constraints. However, this leads to conventional tasks being continuously starved when the system is heavily loaded with real-time applications. Since real-time tasks cannot be preempted by system tasks, a real-time task that does not voluntarily give up the CPU can block out all other system activities, thus, causes the system out of control.

2.2.2 Resource reservation

Resource reservations are commonly combined with real-time scheduling and admission control to support multimedia application tasks in general purpose operating systems [25], [26], [27], [28] and [29]. In order to meet real-time constraints, resource reservation allows real-time tasks to reserve a minimum share of resource that is static and time independent. Once a task reserves a certain share of resource, it is guaranteed to receive at least that share independent of the level of competition for the resource. This rate guarantee is achieved using admission control to reject a new task entering the competing queue if its rate reservation request exceeds the leftover rate available for reservation. EDF allows a total reservation up to 100% of the processor [26] and [27], while RM can only guarantee reservation up to 69% [25]. However, a certain amount of unreserved computation time is required to avoid resource starvation on other tasks. In Rialto [28] the reserved rates of tasks are fixed all along the scheduling, while in [25] and [29] the rate allocations can be modified in a user process

with the support of a monitoring module that monitors task execution and a rate-adaption interface between the kernel and user processes. To avoid tasks receiving CPU cycles more than their processor reservation, a CPU usage monitoring mechanism is required in [25] to measure the CPU usage of each task. However, in [29] such monitoring mechanism can be waived due to the use of scheduling algorithms with a firewall property.

In order to ensure all real-time constraints can be met in applications with dynamic changing workloads, the rates for real-time tasks have to be over-reserved, which leads to task execution rates always falling behind their reserved rates. In this case, a real-time task cannot make use of all its reserved time. At the mean time, any unused CPU time will neither be available for other real-time task reservation, nor for sharing with conventional tasks. The scheduler is non-work-conserving in a way that when a task finishes its execution in a period of time less than its reserved time, the CPU will go to idle even if there are other tasks waiting in the queue to be executed. As a result, conventional tasks will not be able to make use of the slack time left by multimedia applications. Although a rate adaptation mechanism as described in [29] can be utilized to adjust the rates in accordance to the workloads of applications, the required amount of CPU time are difficult to predict since they are both data and hardware dependant, especially in the case of the MPEG video decoder, which is famous for its significant fluctuating workload. Resource reservation based systems are thus better suited when the resource requirements are constant and can be known beforehand. They are not designed to effectively support applications with dynamically changing workloads [30].

Resource reservation relies on admission control to limit system overload. Any new resource request that cannot be satisfied by the leftover CPU capacity will be denied in order to not violate the guarantees to tasks that are already admitted. This achieves fairly strict share guarantee, but in the cost of losing flexibility, fairness and efficiency [30]. With this scheme, a later arriving but more important application may be denied to be allocated resources. Even if the new application is not that important, it is a common situation that a user might be willing to degrade the performance of other applications in order to accommodate a new application [31]. Admission control policy allows a reservation system to shed the system load based on the reservation rate that is allocated in reservation time, thus the system is totally ignorant of the dynamical changing workloads during the program execution. Since over-reserved CPU time

cannot be shared by other tasks, a reservation system might be working in a lightly loaded state and goes to idle regularly, while rejecting new arriving applications.

2.3. Fair queuing

Fair queuing algorithms are widely used in network scheduling [32], [33] and [34] and CPU scheduling [18], [24], [36], [37] and [38]. In the network domain, a fair queuing scheduler ensures different packet flows can fairly share the output link to transmit data with a guaranteed rate. In the CPU scheduling domain, fair queuing algorithms are widely explored to support multimedia and other soft real-time applications on general purpose operating systems, due to their ability to provide strong guarantees to applications and their compatibility with the existing infrastructure of general-purpose operating systems [36] and [37]. This section is divided into two subsections: packet-based fair queuing for network scheduling and time-quantum-based fair queuing for CPU scheduling. In both subsections, the fundamentals of fair queuing are firstly introduced as a theoretical reference to our energy-based fair queuing, and then well-known fair queuing algorithms are discussed and compared.

2.3.1 Packet-based fair queuing

2.3.1.1 Generalized processor sharing (GPS)

The generalized processor sharing (GPS) assumes fluid traffic with infinitesimal packet sizes and serves as an ideal model for packet-based fair queuing algorithms. In a GPS model, a session is any packetized traffic stream that can potentially access a common resource or server. The function of a scheduler or server is to select the packet to be transmitted next from a set of packet queues to access a server. Each session is associated with a reservation rate that determines the share of the capacity or bandwidth of the server that the session is entitled to use. A system busy period is a maximal interval of time during which a server is never idle. A session-i backlogged period is any interval of time during which packets belonging to session i are continuously queued in the system. A session-i busy period is the maximal interval of time during which the arrival rate of session remains or above its reserved rate, so if the session were serviced with exactly the guaranteed rate it would remain continuously backlogged [56].

A GPS scheduler gives service to all backlogged sessions simultaneously and proportionally to their reservation rates. Let ρ_i be the reservation rate of session i, for

any interval $(t_1, t_2]$ during which the set of backlogged sessions does not change, the service received by session i is:

$$W_i(t_1, t_2) = \frac{\rho_i}{\sum_{j \epsilon B(\tau)} \rho_j} \cdot W(t_1, t_2) \quad \forall i \epsilon B(\tau) \tag{2. 1}$$

where $W(t_1, t_2)$ denotes the total service given in the system during $(t_1, t_2]$, and $B(\tau)$ denotes the set of backlogged sessions during that interval. Let $C(t)$ be the server bandwidth at time t, the bandwidth share of session i at time t is:

$$C_i(t) = \frac{\rho_i}{\sum_{\forall j \epsilon B(\tau)} \rho_j} \cdot C(t) \quad \forall i \epsilon B(\tau) \tag{2. 2}$$

The following two conclusions can be further drawn for equation (2.1) and (2.2) :

1. Session i receives at least the guaranteed bandwidth share during any of its backlogged period, independently of the behaviors of any other sessions sharing the server. In other words, for any interval $(t_1, t_2]$ during which session i is continuously backlogged:

$$W_i(t_1, t_2) \geq \rho_i W(t_1, t_2) \tag{2. 3}$$

2. The scheduler gives simultaneously the same normalized service to all backlogged sessions. For any interval $(t_1, t_2]$ during which session i and session j are continuously backlogged:

$$\frac{W_i(t_1,t_2)}{\rho_i} = \frac{W_j(t_1,t_2)}{\rho_j} \tag{2. 4}$$

The performance of a fair queuing algorithm is measured by how close the algorithm can approach a GPS scheduler. It can be quantified by two metrics: fairness and session latency.

The fairness is defined as the difference between the normalized service received by two continuously backlogged sessions i and j over any interval of time $(t_1, t_2]$, represented as [33],

$$\left| \frac{W_i(t_1,t_2)}{\rho_i} - \frac{W_j(t_1,t_2)}{\rho_j} \right| \tag{2. 5}$$

According to equation (2. 4), a GPS scheduler can provide a perfect fairness of zero. A fair queuing algorithm should provide a bounded fairness that is as close to zero as possible.

Session latency is introduced by Stiliadis et al. [39] to represent the worst-case delay seen by the first arriving packet of a session busy period. Since the maximum delay of a packet increases directly in proportion to its session latency, it is important for a fair queuing algorithm to achieve low-latency.

2.3.1.2 The notion of virtual time

The problem of GPS is that it assumes the server can simultaneously serve multiple sessions and that the packets can be transmitted in infinitesimally divisible units. To facilitate the simulation of the ideal model, the constraint of having a perfect fairness can be liberalized to having a bounded service fairness index. Having a non-zero service fairness index means backlogged sessions will not receive the same quantity of normalized service at the same time. However, a bounded service fairness index will guarantee that the difference between the smallest quantity of normalized service and the highest quantity of normalized service is within a bounded variable. Any session that has received the smallest quantity of normalized service at time t should be considered preferentially to receive service, thus, given the opportunity to fairly catch up with the other sessions concerning the quantity of normalized service received.

To evaluate whether or not a session has received at time t the smallest quantity of normalized service in a fair manner, the concept of virtual time, originated from Zhang [40] known as virtual clock, is introduced to record both the missed and received normalized service of one session. The virtual time function is built as follows: while the session is in a backlogged period, the increments of this function measure the normalized service received by the session. On the other hand, when the session is absent from the system in an idle period the function remains unchanged and at the instant of time the session becomes again backlogged, the function is adjusted or updated to add the missed normalized service. Every session has a session virtual time and the system has a system virtual time. For session i, its session virtual time is a function of time $V_i(t)$ that records the quantity of normalized service received up to time, mathematically:

$$V_i(t_2) = \begin{cases} V_i(t_1), & Q_i(\tau) = 0 \ \forall t_1 < \tau < t_2 \\ \max\{V_i(t_2 -), V(t_2)\}, & Q_i(t_2) \neq 0, Q_i(t_2 -) = 0 \\ V_i(t_1) + \frac{W_i(t_1, t_2)}{\rho_i}, & Q_i(\tau) \neq 0 \ \ \forall t_1 < \tau < t_2 \end{cases} \quad (2.6)$$

where $V(t)$ is system virtual time that measures the total normalized service given by the system. $V(t)$ is used to estimate the missed normalized service of one session

during its idle periods and update the session virtual time when it becomes backlogged again. To provide zero waiting time to any newly backlogged session, it should be set to be at any time t less than or equal to the minimum virtual time of all backlogged sessions:

$$V(t) \leq \min\{V_i(t)\}, \forall i \in B(t) \tag{2. 7}$$

Equation (2.7) guarantees a non-decreasing system virtual time function. To provide low session latency and tight delay bound, it is necessary for the system virtual time to have the minimum slope property [41], that is, the system virtual time increases at least with a slope of one.

In a GPS system, system virtual time $V(t)$ is defined to be always equal to the session virtual time $V_i(t)$ of any backlogged session. Hence, for any two sessions i and j that are backlogged at time t , the value of their session virtual time is always the same, $V_i(t) = V_j(t) = V(t)$. Moreover, for any interval $(t_1, t_2]$ during which session i is continuously backlogged:

$$V(t_2) - V(t_1) = V_i(t_2) - V_i(t_1) = \frac{W_i(t_1, t_2)}{\rho_i} = \frac{1}{\rho_i} \cdot \int_{t_1}^{t_2} C_i(\tau)\, d\tau = \frac{W(t_1, t_2)}{\sum_{\forall j \in B(\tau)} \rho_j}$$
$$= \int_{t_1}^{t_2} \frac{C(\tau)}{\sum_{\forall j \in B(\tau)} \rho_j} d\tau \tag{2. 8}$$

and, for a server with constant bandwidth:

$$V(t_2) - V(t_1) = \int_{t_1}^{t_2} \frac{1}{\sum_{\forall j \in B(\tau)} \rho_j} d\tau \tag{2. 9}$$

2.3.1.3 Packet-by-packet GPS (PGPS)

The concept of virtual time provides an effective mechanism in implementing PGPS scheduling algorithms. Originally packet-by-packet implementation of GPS is done in such a way that the packet departing first in the GPS is served first in a real implementation. The time packet n departs from an ideal server is defined as packet finishing time $D_i[n]$ and, in a packet-based scheduler the policy of smallest finishing time first is followed to select the next packet to access the real server. An inconvenience of this policy is that the actual finishing time of a packet depends on the arrival pattern of future packets, thus need to be recalculated each time a packet departs from or arrives to the system. Parekh and Gallager suggested in [32] a practical implementation of PGPS based on virtual time function. The packet finishing

time is replaced by the virtual finishing time to decide the order of packets to be served. Virtual finishing time is defined as the value of session virtual time function at a packet finishing time, which can be determined at the packet arrival time and only changes when there are events in the GPS system. Since the relative order is kept when transforming packet finishing times to their virtual finishing times [42], the policy of smallest finishing time first is equivalent to that of smallest virtual finishing time first (SSF).

To compute the virtual finishing time $\mathrm{FT_i[n]}$ of packet n, it is necessary to know the value of session virtual time in the instant of time when packet n begins to be served and the normalized service received by the session due to the packet n service. The instant of time when packet n begins to be served is defined as the starting time of packet n of session i, $\mathrm{B_i[n]}$, correspondingly the value of session virtual time at $\mathrm{B_i[n]}$ is defined as virtual starting time $\mathrm{ST_i[n]}$. Mathematically:

$$\mathrm{FT_i[n]} = \mathrm{ST_i[n]} + \frac{\mathrm{L_i[n]}}{\rho_i} \tag{2. 10}$$

$$\frac{\mathrm{L_i[n]}}{\rho_i} = \mathrm{V_i(D_i[n])} - \mathrm{V_i(B_i[n])} \tag{2. 11}$$

where $\mathrm{L_i[n]}$ is the packet length of packet n of session i.

Conversely, the virtual starting time can be computed from the virtual finishing time, but two possible situations need to be considered. On one hand, if session i was already backlogged at the arrival instant of packet n, namely $\mathrm{A_i[n]}$, the start time of packet n is equal or greater than the departure time of the previous packet, then in any case the virtual starting time of packet n is equal to the virtual finishing time of the previous packet. On the other hand, if session i is idle before and becomes backlogged at the arrival instant of packet n, the starting time of packet n is identical to $\mathrm{A_i[n]}$. In this case, the virtual starting time is the maximum value between the virtual finishing time of the previous packet and the value of system virtual time function in the packet arrival instant $\mathrm{A_i[n]}$. Mathematically:

$$\mathrm{ST_i[n]} = \begin{cases} \mathrm{FT_i[n-1]}, & \mathrm{Q_i(A_i[n]-)} \neq 0 \\ \max\{\mathrm{FT_i[n-1]}, \mathrm{V(A_i[n])}\}, & \mathrm{Q_i(A_i[n]-)} = 0 \end{cases} \tag{2. 12}$$

$$\mathrm{FT_i[0]} = 0 \tag{2. 13}$$

where $\mathrm{V(A_i[n])}$ denotes the value of system virtual time function at the instant packet n of session i arrives to the system.

From a consideration of efficient computation, equation (2. 14) can be simplified into:

$$ST_i[n] = \max\{FT_i[n-1], V(A_i[n])\} \quad (2.\ 15)$$

By combining equation (2. 16) and (2. 17), finally we have virtual finishing time as following,

$$FT_i[n] = \frac{L_i[n]}{\rho_i} + \max\{FT_i[n-1], V(A_i[n])\} \quad (2.\ 18)$$

Equation (2. 19) provides a recursive mechanism for computing the virtual finishing times. It is necessary to concurrently simulate the ideal GPS model to provide samples of the system virtual time at the packet arrival instants. Therefore, a PGPS scheme may be implemented in the following steps:

1. A packet arrives to its packet queue is stamped with a service tag (as known as time-stamp) that equals to its virtual finishing time.

2. The server serves the packets from packets queues of different sessions in an increasing order of service tag.

2.3.1.4 Packet-based fair queuing algorithms

Since the first packet-based GPS algorithm WFQ was introduced by Demers et al [43], several important fair queuing algorithms for networking scheduling were developed, namely WF^2Q [44], WF^2Q+ [41], SCFQ [33], SFQ [34], VirtualClock [40], FFQ [45] and SPFQ [35]. After that, many following-up fair queuing algorithms such as LFVC [46], MD-SCFQ [47], EFQ [48], SWFQ [49], RP-SCFQ [50], LVT-SCFQ [51], NSPFQ [52] and FPFQ [53] have been proposed. As indicated by the names, they are mainly focused on balancing the fairness, latency and implementation complexity achieved by one of the former algorithms.

According to the analysis in section 2.3.1.1, GPS is considered as an ideal scheduling model with perfect fairness and latency properties. WFQ, by strictly emulating the GPS model, does not fall behind a corresponding GPS system in terms of total service given to each session by more than one maximum size packet [32]. Therefore, WFQ has been regarded as an ideal mean to design a packet-by-packet scheduling algorithm that provides a reference bound of latency and fairness. However, implementation of WFQ needs to concurrently run the simulation of a GPS model. Thus,

WFQ incurs a $O(N)$ complexity in computing the time-stamps for arriving packets, where N is the number of sessions that share the outgoing link. This high complexity in virtual time computation hinders the application of WFQ in high speed networks.

Different algorithms have been proposed to reduce the computing complexity of time-stamps. Self-Clocked Fair Queuing (SCFQ), an approximate implementation of GPS originally proposed by Davin and Heybey [54] and later analyzed by Golestani [33], was designed to reduce the time-stamp computational complexity of WFQ. SCFQ computes times-stamps of arriving packets by only referring to the sessions that are currently backlogged on the actual server, as a results the auxiliary ideal model is no longer needed. Similar to WFQ, packets are stamped with their virtual finishing times and served in the increasingly order of their time-stamps. It has been shown that the fairness bound in a SCFQ scheme is never more than two times the reference fairness bound provided by WFQ. SCFQ employs a virtual time function that updates system virtual time only at the finishing time of any packet and applies an increase slope of zero[1] between the update points, therefore the delay bound can no longer be controlled as in WFQ [35] and [41]. As reported in [55], the end-to-end delay bound of SCFQ can grow linearly with the number of sessions sharing the server. The Start-Time Fair Queuing (SFQ) [34] is similar to SCFQ in the way that a slope of zero is applied between any two update points of the system virtual time. Instead of using the virtual finishing times, SFQ updates the system virtual time from the packet virtual starting times. Correspondingly, packets are scheduled in the increasingly order of their start tags. SFQ achieves a significantly lower delay bound than SCFQ while have the same fairness bound and implementation complexity [34]. However, due to the increase slope of zero introduced between the update points of the system virtual time, the delay bound of SFQ can also grow linearly with the number of sessions sharing the server [34], [35] and [41].

On the other hand, the Virtual Clock (VC) scheduling algorithm, although strictly speaking does not belong to the fair queuing algorithms due to its failure in providing bounded fairness, provides the same delay bound as WFQ [32] and [40]. In VC, the system virtual time is designed as a time function that progresses with the same rate as the real time, which ensures a constant increase slope of one for the updating of the

[1] An increase slope of zero means the virtual time between two update points is constant, the virtual time only changes at the update points.

system virtual time. This is an attractive property. Although VC fails to serve different sessions fairly, it provides a mechanism to simplify the work of updating the system virtual time while achieves a delay bound as low as WFQ. Bennett and Zhang [44] call this property the minimum slope property, and they show it is required for a packet-based fair queuing algorithm to achieve a delay bound that is independent of the number of sessions sharing the server. Similarly, Stiliadis and Varma [56] defines a general class of Rate-Proportional Servers (RPS) that is able to provide the same delay bound as WFQ and substantially variable fairness properties depending on the way the system virtual time is recalibrated to correct any discrepancies. In a RPS server, the system virtual time, known as system-potential function in their work, is defined as a time function that during any interval increases with a rate or slope of at least one and at any instant never exceeds the virtual time of any backlogged session. This is a relaxed definition of the system virtual time of WFQ, which by simulating the GPS model in parallel and tracking the value of service missed for idle sessions at every moment of time, ideally can have the same rate of increase as the virtual time of any session currently being served [35]. It avoids the need of an auxiliary ideal system and provides a mechanism to maintain the system virtual time by approximatively tracking the global state of the system. The general definition of system virtual time in RPS helps to create a framework to design a group of algorithms with the same delay bound as WFQ but different fairness properties that are balanced by their implementation complexities. It is shown that the ideal GPS algorithm, the WFQ and the VirtualClock, all belongs to the RPS class [56]. Based on the RPS framework, two novel scheduling algorithms known as Frame-based Fair Queuing (FFQ) [45] and Staring Potential-based Fair Queuing (SPFQ) [35] were presented. They are the same in the way that an increase slope of one is applied between update points of system virtual time, but differs in how update points are selected to recalibrate the system virtual time. In frame-based fair queuing (FFQ), a framing mechanism is used to select update points to recalibrate the system virtual time periodically by incrementing its value at each update point a fixed quantity T, the frame period. The update points are freely selected within a bounded time interval that depends on the frame size chosen for the implementation. FFQ provides a low delay bound as WFQ and a fairness bound that increases linearly to its frame size. VirtualClock, as a RPS algorithm, can be seen as an FFQ with a frame size of infinity. To improve the fairness bound, Staring Potential-based Fair Queuing (SPFQ), based on the observation of how the fairness of FFQ can vary with the frame size, increases the recalibration frequency of the system virtual time to its extreme by adding update points each time a packet departures the server [35]. The system virtual time at each update point is defined as the minimum of

the virtual starting times of the backlogged sessions, and packets are served in the increasing order of their finishing tags. SPFQ improves the fairness bound of FFQ to a level that is comparable to WFQ and SCFQ, at the cost of a more complex implementation of the recalibration mechanism; however, the asymptotic time complexity keeps the same. The fairness bound of SPFQ and other RPS algorithms can be further improved with a shaping mechanism [57], which releases packets into the scheduler only when the virtual system time becomes equal or greater than the virtual starting time of the packet.

While WFQ is widely believed to be able to provide almost identical service as GPS except for the lag of one maximum size packet, Bennett and Zhang in [44] points out that due to the lack of the control on the advance of sessions, WFQ can be far ahead of GPS with a quantity greater than one maximum size packet, thus leads to a worst-case fairness that may increase linearly to the number of sessions. The Worst-case Fair Weighted Fair Queuing ($\mathrm{WF^2Q}$) scheduling algorithm is proposed to resolve the undesired discrepancy between WFQ and GPS. Unlike WFQ, which selects the next packet to be served among packets of all backlogged sessions, only selects those eligible packets whose virtual starting times are smaller than the current system virtual time, that is, those have started receiving service in the auxiliary GPS system. Correspondingly, the packet selection policy smallest eligible virtual finishing time first (SEFF) is followed to choose the next packet for transmission. By introducing the concept of eligible packets, $\mathrm{WF^2Q}$ prevents WFQ from being far ahead of GPS by more than a fraction of a maximum packet size. Since $\mathrm{WF^2Q}$ also needs to simulate the GPS system, thus it keeps the same time-stamp computational complexity as WFQ. The following-up algorithm, $\mathrm{WF^2Q+}$ [41] provides the same worst-case fairness bound as $\mathrm{WF^2Q}$ with a more efficient implementation that is GPS free. $\mathrm{WF^2Q+}$ has the same system virtual function as SPFQ, but keeps the packet selection policy of smallest eligible virtual finishing time first (SEFF) from $\mathrm{WF^2Q}$. Consequently, $\mathrm{WF^2Q+}$ improves the worst-case fairness of SPFQ. Stiliadis and Varma point out in [57] that $\mathrm{WF^2Q+}$ can be equivalently viewed as a combination of SPFQ with a shaping mechanism.

As an overview, Table 2. 1 lists the fairness, worst-case fairness and latency of the algorithms mentioned above, and Table 2. 2 compares their complexity of time-stamp computation and packets sorting [32], [33], [34], [35], [39], [40], [41], [44] and [45]. All properties are based on a constant service rate. N is the number of sessions sharing the server, $\mathrm{L_i}$ is the maximum packet size of session i and $\mathrm{L_{max}}$ is the maximum packet size among all the sessions. r is the capacity of the server and ρ_i is

the allocated rate of session i. C_i is the maximum normalized service that a session may receive in a PGPS server in excess of that in the GPS server. In frame-based fair queuing (FFQ), F is the frame size.

Algorithm	Fairness	Worst-case Fairness	Latency
GPS	0	0	0
WFQ (PGPS)	$\max\left(C_j + \frac{L_{max}}{\rho_i} + \frac{L_j}{\rho_j}, C_i + \frac{L_{max}}{\rho_j} + \frac{L_i}{\rho_i}\right)$, $C_i = \min\left((N-1)\frac{L_{max}}{\rho_i}, \max_{1\le n\le N}\left(\frac{L_n}{\rho_n}\right)\right)$	$\frac{N}{2}\frac{L_{max}}{r}$	$\frac{L_i}{\rho_i} + \frac{L_{max}}{r}$
WF^2Q	L_{max}	$L_i + (L_{max} - L_i)\frac{\rho_i}{r}$	$\frac{L_i}{\rho_i} + \frac{L_{max}}{r}$
WF^2Q+	L_{max}	$L_i + (L_{max} - L_i)\frac{\rho_i}{r}$	$\frac{L_i}{\rho_i} + \frac{L_{max}}{r}$
SCFQ	$\frac{L_i}{\rho_i} + \frac{L_j}{\rho_j}$	$\geq \frac{N}{2}\frac{L_{max}}{r}$	$\frac{L_i}{\rho_i} + \sum_{1\le n\le N \cap n\ne i}\frac{L_n}{r}$
SFQ	$\frac{L_i}{\rho_i} + \frac{L_j}{\rho_j}$	$\geq \frac{N}{2}\frac{L_{max}}{r}$	$\frac{L_i}{\rho_i} + \sum_{1\le n\le N \cap n\ne i}\frac{L_n}{r}$
Virtual Clock	∞	∞	$\frac{L_i}{\rho_i} + \frac{L_{max}}{r}$
FFQ	$\frac{2F}{r} + \max\left(\frac{L_i}{\rho_i}, \frac{L_j}{\rho_j}\right)$	$\geq \frac{N}{2}\frac{L_{max}}{r}$	$\frac{L_i}{\rho_i} + \frac{L_{max}}{r}$
SPFQ	$\max_{1\le n\le N}\left(\frac{L_n}{\rho_n}\right) + \max\left(\frac{L_i}{\rho_i}, \frac{L_j}{\rho_j}\right) + \frac{L_{max}}{r}$	$\geq \frac{N}{2}\frac{L_{max}}{r}$	$\frac{L_i}{\rho_i} + \frac{L_{max}}{r}$

Table 2. 1 Fairness, worst-case fairness and latency of several fair queuing algorithms

Algorithms	Complexity	
	Time-stamp computation	Packets sorting
GPS	N/A	N/A
WFQ(PGPS)	$O(N)$	$O(\log N)$
WF^2Q	$O(N)$	$O(\log N)$
WF^2Q+	$O(1)$	$O(\log N)$
SCFQ	$O(1)$	$O(\log N)$
SFQ	$O(1)$	$O(\log N)$
VirtualClock	$O(1)$	$O(\log N)$
FFQ	$O(1)$	$O(\log N)$
SPFQ	$O(1)$	$O(\log N)$

Table 2. 2 The complexity of time-stamp computation and packets sorting of several fair queuing algorithms

2.3.2 Time-quantum-based fair queuing

2.3.2.1 The time-quantum-based model and virtual time

Time-quantum-based fair queuing, also known as proportional share scheduling (PSS), aims to emulate a fluid-flow system on a discrete quantum-based allocation system. In the model of proportional share scheduling, a set of tasks or clients run in the system and compete for a shared resource (it refers to the CPU or CPU bandwidth in section 2.3.2). A task can be in two states: active if it is competing for the resource and passive if it is not. Each task T_i is assigned a weight w_i that determines its minimum share of the resource. In an ideal GPS system assuming that multiple tasks can simultaneously receive resource, the share of resource $f_i(t)$ of task T_i at time t is [36]:

$$f_i(t) = \frac{w_i}{\sum_{j \in A(t)} w_j} R(t) \quad (2.\ 20)$$

Where $A(t)$ denotes the set of active tasks at time t, and $R(t)$ denotes the resource allocation rate of the system. If the share of a task remains constant during the interval $(t_1, t_2]$, the task is entitled to use the resource for a time of $f_i(t)(t_2 - t_1)$. If the task share varies along the time, then the service time it receives during any interval $(t_1, t_2]$ is [36]:

$$S_i(t_1, t_2) = \int_{t_1}^{t_2} f_i(\tau)\, d\tau \quad (2.\ 21)$$

From equation (2. 22) and (2. 23), we have:

$$S_i(t_1, t_2) = w_i \int_{t_1}^{t_2} \frac{R(\tau)}{\sum_{j \in A(t)} w_j} d\tau \quad (2.\ 24)$$

In a real implementation of proportional share scheduling, resource is allocated to tasks in the form of discrete time quantum of maximum length Q. A CPU time quantum with a length of Q is called the standard time quantum, that is, the maximum time a task is allowed to continuously use the resource before the next scheduling decision is made. A task is selected to acquire the resource at the beginning of a standard time quantum, it may use resource either the entire standard time quantum or release it before the end of the standard time quantum [36]. This is realized by dividing the service time of task i into smaller pieces of time quantum q_i^k with maximum length Q.

Since the resource allocation is discrete in time, it is impossible for a task to receive exactly the same service as it is entitled to in the ideal GPS system. As a result, an allocation error is generated, which can be measured by the unfairness as defined in the network scheduling domain, that is, the difference between the normalized service received by two tasks over an interval of time during which both are continuously active. Given two tasks T_i and T_j that are continuously active during an interval $(t_1, t_2]$, and let $s_i(t_1, t_2)$ denotes the actual service time task i has received during the interval $(t_1, t_2]$, the allocation error can be defined as follows [58],

$$\epsilon_{i,j} = \left| \frac{s_i(t_1, t_2)}{w_i} - \frac{s_j(t_1, t_2)}{w_j} \right| \quad (2.\ 25)$$

A proportional share scheduling algorithm should minimize the resultant unfairness and ensure it as close to zero as possible [37].

Stoica in [36] introduce a more practical method named service time lag to measure the allocation error. A lag is defined as the difference between the service time a task should receive in an ideal GPS system and the service time it actually receives in the real system. Mathematically, let t_0^i be the time when task i becomes active, let $s_i(t_0^i,\ t)$ be the actual service time the task receives during interval $(t_0^i,\ t]$, and $S_i(t_0^i,\ t)$ be the ideal service time the task should receive during that interval, then, the service time lag of task i at time t is [36]:

$$lag_i(t) = S_i\left(t_0^i,\ t\right) - s_i(t_0^i,\ t) \quad (2.\ 26)$$

The lag quantifies the allocation error [36]. Proportional share algorithms should be designed to have bounded lag in order to support time-sensitive tasks in time-shared operating systems [58]. A positive lag indicates that a task in implementation has received less service time than the service time received in the ideal GPS system, and a negative lag indicates a task has received more service time than the service it should have received.

As in the network scheduling domain, the concept of virtual time is introduced to implement proportional share scheduling in a more efficient manner. The system virtual time is defined as:

$$V(t) = \int_0^t \frac{R(\tau)}{\sum_{\forall j \epsilon A(\tau)} w_j} d\tau \quad (2.\ 27)$$

For a system with constant resource allocation rate, the system virtual time is [37]:

$$V(t) = \int_0^t \frac{1}{\sum_{\forall j \epsilon A(\tau)} w_j} d\tau \quad (2.\ 28)$$

By combining equations (2. 29) and (2. 30), the service time received by an active task i during an interval $(t_1, t_2]$ can be expressed as [36],

$$S_i(t_1, t_2) = w_i(V(t_2) - V(t_1)) \quad (2.\ 31)$$

Similarly, each time slice q_i^k is assigned a virtual starting time $S(q_i^k)$ and a virtual finishing time $F(q_i^k)$, defined as the value of the task virtual time at the instant of time when the time slice q_i^k begins execution and finishes execution, respectively, computed as[34]:

$$S(q_i^k) = \max\left(V\left(AR(q_i^k)\right), F(q_i^{k-1})\right) \quad (2.\ 32)$$

$$F(q_i^k) = S(q_i^k) + \frac{q_i^k}{w_i} \quad (2.\ 33)$$

where $AR(q_i^k)$ denotes the instant of time at which the k^{th} time quantum of task i is requested. If the time quantum q_i^k is requested at the moment that task i is making a transition from passive state to active state, then $AR(q_i^k)$ equals to the time at which the transition is made; otherwise, $AR(q_i^k)$ equals to the moment when the previous quantum q_i^{k-1} of task i finishes execution [38].

2.3.2.2 Share protection for time-sensitive tasks

Proportional share scheduling algorithms aim to support real-time tasks in general-purpose time-sharing systems. Thus, a fixed, minimum share of CPU bandwidth is required for time-sensitive task. However, according to equation (2. 16), the CPU bandwidth share f_i of task T_i varies with the number and total weight of the tasks in the system. Any new task joins the competition with a large weight may reduce the CPU share of a time-sensitive task to a arbitrarily low level and lead to significant deadline missing. To protect and fix the CPU share of a time-sensitive task, its weight must vary with respect to other task weights when tasks join or leave the system. Goddard and Tan called this the weight-assignment problem [59].

Waldspuerger and Weihl firstly provide mechanisms to achieve resource share protection (known as load insulation in their work) among different class of tasks in a probabilistic proportional-share algorithm termed lottery scheduling [60]. Lottery scheduling uses tickets to abstractly specify resource rights, and provides supports for modular resource management with multiple ticket currencies. A currency defines a resource management abstraction barrier within which the impact of currency fluctuation such as ticket inflations is localized. The currency abstraction can be used to flexibly isolate different group of users and protect their resource rights. However, Waldspuerger and Weihl did not give out how load insulation is implemented in lottery scheduling, thus the implementation and time complexity is unknown.

Stoica et al. [31] explore the duality of weights and shares in proportional share scheduling, and propose a weight re-computation method to guarantee stable shares to real-time tasks. A task is characterized simultaneously by its weight and share, that is (w_i, f_i), where w_i represents the weight for competing the CPU bandwidth, and f_i represents the actual share of CPU bandwidth received by one task. Suppose the sum of shares of all the tasks is one, the share of a task can be defined as,

$$f_i = \frac{w_i}{W} \tag{2. 34}$$

where W is the total weight of all active tasks in the system. Alternatively, if a task requires a share f_i of CPU bandwidth, the weight required by a task can be computed as,

$$w_i = \frac{f_i\,(W - w_i)}{1 - f_i} \tag{2. 35}$$

By fixing the weight w_i, a non real-time task can receive a CPU share that is proportional to value of the weight relative to the total weight of all active tasks in the system; by fixing the share f_i, a real-time task can re-compute its weight w_i by equation (2. 27) to achieve a constant resource reservation for supporting real-time execution.

This method has been applied to a proportional share scheduling algorithm named EEVDF [36], and results show that the CPU share of a real-time task can be fixed and not affected by dynamic task participation. However, it was pointed out by Goddard and Tan [59] that the recalculation of w_i could be quite complex because for n real-time tasks n equations need to be resolved. In response, Goddard and Tan developed a simpler method for weight recalculation. As in [31], tasks are separated into two classes, real-time class and non real-time class. The total weight of all tasks

equals to one ($\sum_{i\in A} w_i = 1$). Each time the task number changes, the weight of a real-time class task is fixed to its desired share $w_i = f_i$, and the share left by real-time class is reallocated to non real-time tasks based on their recalculated weights. Let F be the total share reserved to k tasks of real-time class, then,

$$F = \sum_{i=1}^{k} f_i \tag{2. 36}$$

and the weight of a non real-time task is recomputed as,

$$w_i = \frac{\overline{w}_i}{\sum_{j=k+1}^{n} \overline{w}_j}(1 - F) \tag{2. 37}$$

where $\overline{w}_i$ is the original weight of a non real-time task. In this way, whenever a task joins or leaves the system, the power shares of time-sensitive tasks remain unchanged whereas the power shares of non real-time tasks are reallocated.

Recently, a new mechanism [61] based on the EEVDF algorithm was proposed to flexibly support a mix of applications with various resource requirements. In this mechanism, each task has an initial share $\bar{f}_i \in [0,1]$, and an initial weight $\overline{w}_i$, based on which tasks are classified into three different categories [61]:

- $\bar{f}_i = 0$ and $\overline{w}_i > 0$, for regular tasks do not require a guaranteed CPU bandwidth share for meeting time-constraints.
- $\bar{f}_i > 0$ and $\overline{w}_i = 0$, for these time-sensitive tasks that only ask for a guaranteed share, and not compete for unreserved or any released CPU bandwidth in the system.
- $\bar{f}_i > 0$ and $\overline{w}_i > 0$, for these time-sensitive tasks that not only require a guaranteed share but also competes for unreserved or any released CPU bandwidth with their initial weight.

Based on the values of initial share and initial weight, an effective share f_i and effective weight w_i is calculated for each task. Then proportional share scheduling algorithms such as EEVDF can use effective weight for computing virtual times and making scheduling decisions. The effective share is computed as the addition of a task´s initial share and its share of free CPU bandwidth, and the effective weight is computed in accordance to the effective share. In overload state when the sum of the initial shares of all the tasks exceeds 100% ($\sum_{j\in A(t)} \bar{f}_j \geq 1$), the effective share equals to

its initial share $f_i = \bar{f}_i$, and the effective weight is $w_i = \bar{f}_i$. In underload state when the sum of initial shares is less than 100% ($\sum_{j \in A(t)} \bar{f}_j < 1$), the effective share is,

$$f_i = \bar{f}_i + \left(\frac{\bar{w}_i}{\sum_{j \in A(t)} \bar{w}_j} \right) \cdot (1 - F),\ F = \sum_{j \in A(t)} \bar{f}_j \qquad (2.\ 38)$$

and the effective weight is,

$$w_i = \frac{\bar{f}_i \cdot \sum_{j \in A(t)} \bar{w}_j}{1 - \sum_{j \in A(t)} \bar{f}_j} + \bar{w}_i \qquad (2.\ 39)$$

According to equation (2. 31), for a regular task whose initial share is zero, its effective weight equals to its initial weight $w_i = \bar{w}_i$, therefore no recalculation is needed. However, for any time-sensitive task with a non-zero initial share, the effective weight has to be recalculated by using equation (2. 31). As a result, this mechanism is better suited for systems with a small number of time-sensitive tasks.

2.3.2.3 Proportional share scheduling algorithms

There are several compelling reasons that make proportional share scheduling (PSS) a good candidate to support multimedia and other soft real-time applications on general-purpose operating systems [36] and [62]. First, PSS characterizes different types of task with the same parameter, a share. Thus, PSS is able to seamlessly integrating real-time and non-real-time tasks. Second, compared to traditional time-sharing scheduler, a PSS scheduler provides a stronger performance guarantee to time-sensitive tasks. Third, PSS provides a natural means to restrict the resource usage of ill-behaved or high-demanding tasks, and protect well-behaved tasks from resource starvation. Forth, a PSS scheduler allows graceful degradation of system performance in overload situation. Finally, PSS can be easily implemented due to its compatibility with the existing infrastructure of general-purpose operating systems.

The rationale of PSS in supporting real-time execution is based on the proportional share of CPU bandwidth. In the traditional real-time scheduling model, a periodical task is guaranteed to receive a certain amount of CPU time during each period. This can be viewed as a coarse approximation of a CPU bandwidth share in the PSS. In an ideal fluid-flow system, PSS can meet all the time constraints if the resource requirements of tasks are no greater than their respective assigned shares. However, setting proper PSS shares for periodical tasks in a real system is a difficult work because it requires taking the error bound as well the dynamically changing resource requirements into account. For a task with actual execution time a, worst-case

execution time (WCET) c and period p running in a PSS scheduler with error bound lag, meeting all the time constraints requires at least a share s of $(c + lag)/p$. If the PSS share is adaptive, a variable share $(a(t) + lag)/p$ is required to meet all the time-constraints, $a(t)$ means the value of the actual execution time varies with time. However, in most multimedia and soft real-time applications, the actual execution time a varies in different periods and is hard to predict. To avoid the overhead of prediction as well as any missed deadline caused by prediction error, the share can be simply fixed to be $(c + lag)/p$. In both cases, the CPU time is over-reserved for periodical tasks. Regehr in [62] use the pessimism to reflect the over-reservation. The pessimism P is defined as the amount of CPU time reserved for a task divided by the actual execution time, computed as sp/a. In the case of adaptive share, the pessimism is $1 + lag/a$; in the case of fixed share, the pessimism is $(c + lag)/a$. In a real system, the pessimism is always larger than one. It is noticeable that in the later case, the pessimism can be a considerable value if WCET c is much larger than the actual execution time a. As the total share of all tasks is one ($\sum_{i \in A(t)} f_i = 1$), a share with $P > 1$ causes a waste of CPU time allocation in the sense that the over-reserved CPU time cannot be guaranteed to other tasks [33]. Generally, two research approaches have been followed to resolve the pessimism problem. The first approach is focused on reducing the value of pessimism. Since the actual execution time a and worst-case execution time c both depend on the given real-time task, the only way to reduce the pessimism in both cases is to minimize the allocation error lag. Proportional share scheduling algorithms such as EEVDF [36] are designed to have the optimal fairness bound as well as error bound. The second approach combines extra real-time friendly mechanisms into the PSS, aiming to provide better real-time performance while allowing shares to be assigned in accordance with user desired allocation. Examples can be found in algorithms like SMART [24], BERT [63] and BVT [38]. Those algorithms, however, improve the real-time performance at the cost of fairness. Besides of the fairness and allocation error bound, proportional share scheduling algorithms are also evaluated by their implementation complexity and run-time efficiency [64].

In the remaining part of this section, we first analyze why WFQ fails when directly applied to the CPU scheduling. Following that, we discuss and compare several well-known proportional share scheduling algorithms, based on the fairness, allocation error bound, real-time performance, implementation complexity, and run-time efficiency.

2.3.2.3.1 Why WFQ fails

With the time-quantum-based model, fair queuing scheduling algorithms can be cross-applied to the CPU scheduling domain to achieve proportional share of CPU among different tasks. A direct approach for implementation is the WFQ. As has been introduced in section 2.2.3, WFQ schedules tasks in the increasing order of virtual finishing time, thus, it needs a priori knowledge of the length of a time slice. In network scheduling, the length of a packet is able to be known from the header when the packets arrives [65]. However, in CPU scheduling it is usually impossible to predict the amount of service time that will be actually used by a task [66]. As a result, it needs to assume that each time slice equals to the maximum size Q, which leads to a task receiving an unfair share if less service time is used by the task. Also, WFQ does perform well in dynamic task participation. In CPU scheduling, if a task is allowed to dynamically join or leave the system at any time, it may leave the competition before being allocated a quantum of service time or just after using up one time quantum. This introduces unfairness that can be reflected by the difference value between the task virtual time ($task\_pass$) and system virtual time ($global\_pass$), or by a non-zero lag. This problem does not occur in network session scheduling with constant rate since the size and transmission time of a packet is known upon its arrival, thus, has not been addressed in WFQ. Moreover, due to the discrepancy between WFQ and GPS, a task can has a positive lag that increases in proportion to the number of active tasks in the system [41] and [44]. Last but not least, since WFQ needs to run the auxiliary GPS system for frequently recalculating the system virtual time $V(t)$, the runtime overhead of CPU is increased.

2.3.2.3.2 Stride scheduling

Stride scheduling [18] is the first fair queuing algorithm that was developed and implemented for CPU scheduling. The basic idea of stride scheduling is similar to WFQ. In stride scheduling, every task is holding a ticket, which specifies its resource allocation relative to the total number of tickets in the system. The $stride$ is inversely proportional to the tickets and represents the time interval that a task must wait between successive allocations. The smaller $stride$ one task has, the more frequent it will be scheduled. The $stride$ is measured in virtual time units named passes (similar to virtual time in WFQ), which records the work progress of tasks. A $global\_pass$ (system virtual time) is available to record the total normalized service given in the system, so that to update the $task\_pass$ (task virtual time) of any newly-joined task. As in WFQ, the next task to be allocated one time quantum is the one with the lowest finishing pass.

Stride scheduling can be seen as a cross-application of WFQ to the CPU scheduling, it deals with the following issues caused by directly applying WFQ:

- Supports fractional and non-uniform quanta. Although a constant time quantum named stride_1 is used to compute the strides for all tasks, stride scheduling provides mechanisms to support non-uniform quanta if a task consumes less than the standard allocated time quantum Q. This is realized by advancing the pass by $f \times \text{stride}$ instead of stride, where f represents the fraction of the standard allocated time quantum that is actually consumed by a task and is computed as the elapsed resource usage time divided by Q. Stride scheduling also allows task to specify its required quantum size, deviations from a task´s specified quantum can be handled with the same method as described above.

- Addresses the problem of maintaining fairness in dynamic task participation. In stride scheduling, a state variable named remain is defined to store the number of passes that are left before a task´s next dispatch; when a task rejoins the system, its task_ pass value is recalculated by adding to the global_pass its remain value. If $\text{remain} < \text{stride}$, the task is favored to receive service when it rejoins for "having previously waited part of its stride without receiving a quantum"; if $\text{remain} > \text{stride}$, then the task is punished when it rejoins for "having previously received a quantum without waiting its entire stride" [18]. This approach introduces a time complexity of $O(\lg n_c)$ for both operations regarding task leave and rejoin, where n_c is the number of active tasks in the system.

- Supports dynamic modification of relative allocations. If a task´s allocation is changed from tickets to tickets´, both its stride and task_pass have to be recomputed. As usual, the new stride´ is inversely proportional to the tickets´. The new task_pass´ is computed by adding the new remain´ to the global_pass, where the new remain´ is recomputed by scaling the remain by $\text{stride´}/\text{stride}$. The above modification requires a time complexity of $O(\lg n_c)$, where n_c is the number of active tasks in the system.

- Although not stated in [18], stride scheduling can also support resource share protection by grouping tasks in different currencies.

Stride scheduling provides a solution to the fix-quanta problem of WFQ, but at the end of the execution of each time quantum, the finishing tag has to be modified to reflect the actual length of the execution, which introduces extra overhead to the algorithm. Similar to WFQ, stride scheduling allocates one time quantum to the task with the lowest finishing tag, therefore is claimed to have a $O(n_c)$ lag, where n_c is the number of active tasks in the system. However, it is not known if with the above modification stride scheduling still retains this fairness property due to the lack of theoretical proof. Moreover, although the lag can be reduced to a tighter $O(\lg n_c)$ bound by "grouping the tasks in a binary tree, and recursively applying the basic stride scheduling at each level" [18] and [36], it can be still considerable when there are a large number of active tasks in the system. Stride scheduling uses the $remain$ to maintain the fairness in dynamic task participation. This approach is based on the assumption that a current partial quantum is equivalent to a partial quantum in the future [18]. However, if the tasks competing for the resource vary significantly between the time that a task leaves and rejoins the system, extra unfairness will be incurred [18] and [66]. For example, later joined new tasks that are not active in the task leaving time will be unfairly given credits or penalized in receiving service time. Finally, stride scheduling also need to relay on a GPS system to update the system virtual time ($global\_pass$).

2.3.2.3.3 Earliest eligible virtual deadline first (EEVDF)

Stoica et al. in [36] proposes the Earliest Eligible Virtual Deadline First (EEVDF), a proportional share resource allocation algorithm aiming to support real-time tasks in general purpose operating systems. In order to realize real-time performance in time-shared operating systems, EEVDF guarantees the optimal lag bound Q, that is, the difference between the service time that a task should receive in an ideal GPS system and the service time it actually receives in a real system is no more than one time quantum. EEVDF defines the virtual time as in WFQ and schedules tasks according to their virtual finishing times (called virtual finishing deadline in [36]). However, in order to avoid the $O(n_c)$ positive lag, only those tasks whose virtual starting time (called virtual eligible time in [36]) are smaller than the system virtual time is considered as eligible to be scheduled. This means, a task (or its time quantum request) becomes eligible at the instant when the service time that the task should receive in the ideal GPS system equals the service time it has already received in the real system. Consequently, "tasks that have received more service time are slowed down, giving other active tasks the chance to catch up" [36].

As in stride scheduling, EEVDF addresses the problem of maintaining fairness in dynamic task participation, but in a more systematical and argumentative way. The fairness problem does not occur as long as all tasks leave the system with zero lag. A simple solution to reach that is to consider the leaving time of a task as its leaving time in the corresponding GPS system, in which the lag of any task is always zero. Unfortunately, this solution need expensive overhead in maintaining the dynamic events in GPS system and, more important, implicitly assumes the service time to be known in advance [36]. In EEVDF, the total lag of all active tasks at any time is zero. Therefore, a task leaves the system with a positive lag (receive less service than it should receive) will lead to the remaining tasks receiving more service. Similarly, if a task leaves with a negative lag, the remaining tasks will receive less service. The lag generated by one leaving task is proportionally distributed among the rest active tasks in accordance to their weights. Based on this observation, EEVDF proposes a solution with the following three steps:

1. Only tasks with non-negative lags are allowed to leave the system. Any task with a negative lag that wants to leave the competition is simply delayed (without being allocated any service time since it has already finished using the resource) until its lag becomes zero. In this way, only the case of positive lag is considered.

2. A task with a positive lag is allowed to leave the system immediately. To proportionally distribute its lag to the remaining active tasks, the system virtual time is updated as:

$$V(t) = V(t) + \frac{lag_j(t)}{\sum_{i \in A(t) \setminus \{j\}} w_i} \quad (2.40)$$

   where $\sum_{i \in A(t) \setminus \{j\}} w_i$ denotes the sum of all active tasks just after task T_i leaves the system.

3. The lag of a previously leaved task is not preserved, all tasks that join or rejoin the competition are assumed to have zero lag.

EEVDF deals with the fairness problem in dynamic task participation without incurring extra unfairness as in stride scheduling. On one side, it maintains the fairness in dynamic system by proportionally distributing the lag of a leaving task to the remaining tasks; on the other side, since all tasks join or rejoin the system with a zero lag, no extra unfairness will be introduced if there is significant task variation between

the task leaving time and task rejoining time. Besides, since in step 2 system virtual time is updated according to the dynamic events actually occur in the real system, EEVDF can be easily and efficiently implemented without keeping an event queue in the ideal system. However，an ideal GPS system is required to frequently update the system virtual time and compute the lag of a leaving task. Thus, the time complexity of computing time stamps is $O(n_c)$, where n_c is the number of active tasks in the system. EEVDF can be seen as a cross-application of WF^2Q to the CPU scheduling. It modifies the virtual eligible time of the next quantum according to the actual length of the former quantum, thus, also supports fractional and non-uniform quanta. However, different from stride scheduling, EEVDF gives theoretical results showing that the constant fairness bound achieved in WF^2Q can still be retained in CPU scheduling. Actually, EEVDF provides the theoretical foundations that are required to achieve real-time performance in time-sharing general-purpose operating systems. Besides the $O(n_c)$ time complexity in time stamp computation, the time complexity of implementing other basic operations in EEVDF is $O(\lg n_c)$, where n_c is the number of active tasks in the system.

2.3.2.3.4 Starting-time fair queuing (SFQ)

Starting-time fair queuing (SFQ) was first proposed by Goyal for network scheduling [34] and later applied to the CPU domain for hierarchically portioning of CPU bandwidth among different application classes [37]. As the name indicates, SFQ schedules tasks in the increasing order of their virtual starting times, from which the system virtual time is updated without referring to an ideal GPS system. Compared to WFQ, SFQ provides the following attractive properties [34] and [37]:

- Computationally efficient. As we know, WFQ needs to simulate the ideal GPS system to compute the system virtual time, which is computationally expensive either for high rate network scheduling or for processor scheduling. In SFQ, the system virtual time at time t, $V(t)$, is defined to be equal to the virtual starting time of the task in service at time t. Thus, system virtual times can be updated by referring to the real system itself without incurring the high computation overhead of maintaining the ideal GPS system. The time complexity for computing the starting virtual times in SFQ is $O(1)$, and for sorting the service quanta is $O(\lg n_c)$, where n_c is the number of active tasks in the system.

- Not need to know the length of a service quantum in advance. Since SFQ decides the scheduling order based on the virtual starting time, the length of time quanta is not required at the instance of scheduling. As a result, SFQ supports fractional and non-uniform quanta as in stride scheduling and EEVDF. Different from stride scheduling and EEVDF, no extra overhead is incurred for modifying the advanced task virtual time when finishing one service quantum. This property is highly appreciated in multimedia task scheduling, where the service time may vary dramatically and hard to predict.

- Provides fairness bound under fluctuating bandwidth. The fairness bound provided by WFQ (also stride scheduling and EEVDF) is under the condition that the available network or CPU bandwidth is constant. Goyal shows in [34] that WFQ fails to provide its claimed fairness bound when the CPU bandwidth fluctuates over time. On the contrary, SFQ is able to provide fairness guarantee regardless of the fluctuations in system processor bandwidth. For any interval $(t_1, t_2]$ during which two tasks m and n are continuously active in the system, the differences in the service time received by two tasks is:

$$\left|\frac{s_n(t_1,t_2)}{w_n} - \frac{s_m(t_1,t_2)}{w_m}\right| \leq \frac{l_n^{max}}{w_n} + \frac{l_m^{max}}{w_m} \qquad (2.\ 41)$$

  where l_i^{max} denotes the maximum length of quantum of task i. This is a near- optimal fairness bound in reference to the one achieved by WFQ.

- Provides throughput (share) guarantee and bounded delay under fluctuating bandwidth that can be modeled as a Fluctuation Constraints (FC) server [67] or Exponentially Bounded Fluctuation (EBF) server [67]. FC server is defined as a lower bounded server that has a long-term average data rate C and a burstiness $\delta(C)$, and in any interval of a busy period transmits at least δ(C) less data than a corresponding constant rate server. An EBF server is a stochastic relaxation of FC server. For a processor that can be modeled as a FC server with parameters $(C, \delta(C))$, the resource share actually received by a task i is also FC with parameters:

$$(w_i, w_i \frac{\sum_{n \in Q} l_n^{max}}{C} + w_i \frac{\delta(C)}{C} + l_f^{max}) \quad (2.\ 42)$$

and for a service quantum q_m^j of task m that is expected to arrive at time $EAT(q_m^j)$, its completion time, denoted by $L_{SFQ}(q_m^j)$, is given as,

$$L_{SFQ}\left(q_m^j\right) \leq EAT\left(q_m^j\right) + \sum_{i \in Q \wedge i \neq m} \frac{l_i^{max}}{C} + \frac{l_m^j}{C} + \frac{\delta(C)}{C} \quad (2.\ 43)$$

- SFQ provides lower delay when the number of active tasks is small, especially for low throughput (share) applications. According to the latency of WFQ shown in Table 2.1, WFQ guarantees a service quantum q_m^j of task m to be completed by time,

$$EAT\left(q_m^j\right) + \frac{l_m^j}{w_m} + \frac{l^{max}}{C} \quad (2.\ 44)$$

where l^{max} is the maximum length of quantum among all tasks. According to equation (2. 45) and (2. 36), the difference between the maximum delay incurred in SFQ and WFQ under constant bandwidth, denoted by $\Delta(q_m^j)$, is,

$$\Delta\left(q_m^j\right) = \sum_{i \in Q \wedge i \neq m} \frac{l_i^{max}}{C} + \frac{l_m^j}{C} - \frac{l_m^j}{w_m} - \frac{l^{max}}{C} \quad (2.\ 46)$$

$\Delta\left(q_m^j\right) < 0$ when the number of tasks or w_m is small. Thus, lower delay is provided by SFQ, especially for low throughput (share) tasks, such as interactive tasks.

SFQ is simple and efficient to implement, and provides bounded fairness, throughput and delay. However, the simplicity and high efficiency does not come for free. As implicitly indicated by the delay bound in equation (2. 35), the lag bound of SFQ increases linearly with the number of active tasks. Besides, since the system virtual time in SFQ is roughly approximated according to the virtual starting time of the task in service, SFQ fails to provide any method to maintain fairness in dynamic task participation.

2.3.2.3.5 SMART: a scheduler for multimedia applications

Nieh and Lam proposed the SMART [24] to support multimedia and real-time applications on general-purpose operating systems. The crux of SMART is to distinguish between urgency and importance when making scheduling decisions. While urgency is specific to real-time tasks and measured by the time constraints, importance is common to all applications and measured by a value-tuple consisting of a static priority and a biased virtual finishing time (BVFT) that reflects the normalized energy received by a task. In the computation of BVFT, a bias is added to the virtual finishing time of regular tasks when completing a quantum, so that regular tasks are deferred to let real-time tasks get scheduled earlier to meet their time constraints. The bias affects instantaneous proportional allocations and worsens fairness bound, but does not change the long-term proportional share of resources.

SMART makes the scheduling decision in two steps: the first identifies all the tasks that are considered important enough to be executed, and the second chooses among the important tasks the most urgent one to be executed. A more detailed description of the steps followed to select the next task is given as follows:

1) If the task with the highest value-tuple is a regular task, schedule that task immediately.

2) Otherwise, create a candidate set of all the real-time tasks that have a higher value-tuple than that of the highest ranked regular task.

3) Order the tasks of the candidate set by using the value-tuple as the priority, and then apply EDF to insert a task into a working schedule only on condition that its execution does not cause any higher priority task to miss its time constraint.

4) Schedule the tasks in the working schedule by using EDF.

Compared with traditional proportional share scheduling algorithms like EEVDF, SMART introduces time constraint awareness to the scheduler, thus, provides better real-time performance while allowing proportional sharing of resource based on user desired allocations. In addition, SMART integrates static priority into the proportional share scheduling and allows prioritize tasks across real-time and non-real-time classes. Besides, SMART provides dynamic feedback to real-time applications to allow them adapt properly to the current load.

However, SMART experiences several issues in implementation. First, SMART incurs cost in managing both the value-tuple list and the working schedule. Since system virtual time is updated in reference to a GPS system and tasks are ordered based on the virtual finishing time, the cost of managing the value-tuple list is similar to WFQ. It requires a time complexity of $O(n_c)$ for finishing tag computations and $O(\lg n_c)$ for service quanta sorting. The complexity of managing the working schedule is $O(n_R^2)$, where n_R is the number of real-time tasks in the candidate set. This complexity can be further reduced to $O(n_R)$, but requires a more complicated implementation scheme. Second, SMART predicts the service time required by a periodical real-time task in its future periods, tests if the required service time can be served before its deadline, and based on that decides if insert a task into a working schedule or abandon its service request. This method is sensitive to the prediction error, thus, introduces some risk that a service request is abandoned when actually it can meet its deadline, and some risk of converse. Third, SMART introduces a bias to the virtual finishing time of regular tasks to preferentially schedule real-time tasks, but does not provide any application to support flexible control of this bias.

2.3.2.3.6 Borrowed-virtual-time (BVT) scheduling

Duda and Cheriton proposed the Borrowed-Virtual-Time (BVT) [38] scheduling, an effective yet low-complexity algorithm that supports time-sensitive tasks in general-purpose operating systems. The scheduling behavior of BVT is similar to SFQ, however, a real-time friendly mechanism named warping is employed to support time-sensitive tasks. The warping mechanism involves the following state variables of each task: A_i, the actual virtual time (AVT); E_i, the effective virtual time (EVT); W_i, the virtual time warp; and $\mathrm{warpBack}_i$, a bool sets if warp is enabled. The effective virtual time is computed as,

$$E_i = \begin{cases} A_i - \mathrm{warp}_i, & \mathrm{warpBack}_i = 1 \\ A_i, & \mathrm{warpBack}_i = 0 \end{cases} \quad (2.47)$$

BVT monitors the task execution progress with the actual virtual time, but schedules tasks in the increasing order of the effective virtual time. By enabling the warp to warp back the virtual time stamp, a time-sensitive task appears earlier in the scheduling queue and gains dispatch preference. The fairness worsens due to the dispatch preference given to time-sensitive tasks; however, the long-term CPU bandwidth share is still constrained by the weighted fair sharing of BVT, because the actual virtual time A_i is advanced based on its actual CPU usage.

Warping a task can introduce latency to other lower-priority tasks. Thus, BVT introduces two additional parameters to warping: the warp time limit L_i that limits the maximum time one task can run warped; and the unwarp time requirement U_i that governs the time a task must wait before warping again. These two warp parameters can be properly set to limit the CPU occupation of higher-priority tasks and thus avoid adding too much latency to other tasks. BVT provides a user interface to support flexibly setting of these two parameters.

The warping mechanism is simple and straightforward to implement in any WFQ-like fair-queuing algorithm. With proper choice of warp parameters, BVT can reduce latency for real-time and interactive tasks while providing weighted sharing of the CPU bandwidth across time-sensitive and regular tasks. However, since the interaction of multiple warped tasks has not been quantified and it is still an open question how various warp parameters should be set to produce a desired overall system behavior, more researches on the warping mechanism are required [63].

2.4. Summary

In this chapter, we have made a review on research works related to lifetime-oriented power management schemes, real-time scheduling algorithms and fair queuing scheduling algorithms.

Lifetime-oriented power management schemes are designed to achieve a target lifetime for mobile systems. To achieve that, some of them rely on the cooperation of energy-aware applications to self-adapt their energy demand according to the energy state of the system. The problem is that they require the applications to be self-adaptive and the lifetime extension space is limited by the level of quality degradation supported by applications. The energy-centric scheme provides a wider lifetime guarantee by globally managing energy as the first-class resource in the OS. It supports general applications without requiring them to be energy-aware. However, time-constraint meeting are not considered in this scheme, and applications that are forced to idle before the end of epoch will suffer from losing consistency. It is suggested that the energy-centric scheme can be combined with application adaption and energy-efficient algorithms such as DVFS to form more sophisticated PM schemes.

Real-time scheduling algorithms such as rate-monotonic (RM) and earliest deadline first (EDF) are priority-based scheduling algorithms that provide strict time-constraint meeting in under-loaded systems. To guarantee the time-constraint meeting

in general purpose operating systems, real-time tasks have to be assigned strictly higher priority than other tasks, which brings the risk to resource starve the low-priority tasks or even leads to the system out of control. Resource reservations are commonly combined with real-time scheduling and admission control to reserve a minimum amount of computational time for guaranteeing the time-constraint meeting of any real-time task. Depends on the real-time scheduling algorithm employed, the admission control allows a total reservation up to a certain level, thus leaves a amount of unreserved computational time to avoid resource starvation on other tasks. However, the scheduler based on resource reservation is non-working-conserving in a way that any over-reserved and unused resource will neither be available for other real-time task reservation, nor for sharing with regular tasks. Make it worse, to protect the already reserved resource, the admission control may deny a later coming but more important task while the system is working in a lightly loaded state.

Fair queuing has been widely applied in network scheduling and CPU scheduling to allow different entities fairly and proportionally share the resource according to their assigned shares. It is work-conserving in a way that any reserved but unused resource can be shared by other tasks. Fair queuing is based on the fluid-flow system in which resource is assumed to be infinitely divisible and can be simultaneously served to different tasks. While a packet-based fair queuing scheduling algorithm is designed to approach the fluid-flow system with the minimum fairness and latency bound, a time-quantum based fair queuing scheduling algorithm concerns more about the resource allocation error, which may delay the execution of a time-sensitive task in refer to the fluid-flow system and thus, brings risk for time-constraint meeting. A fair queuing algorithm treats all tasks as one type and supports time-constraint meeting by allocating any time-sensitive task a proper share that is adequate to meet its worst-case workload. However, computing the minimum share for time-constraint meeting requires taking the allocation error into account, which leads to the requirement of an over-reserved share. The over-reserved resources are wasted in the sense that they cannot be reserved to other time-sensitive tasks. To minimize the over reservation, algorithms like EEVDF are designed to achieve the optimal allocation error that is no larger than one time quantum. However, their high overhead in supporting non-uniform quanta and maintaining reference fluid-flow system prevents their application in a practical scheduler. On the other side, low-overhead algorithms like SFQ are more practical to implement, but the allocation error can increase linearly to the number of active tasks in the system. Based on this observation, real-time friendly algorithms are combined into fair queuing to better support time-sensitive tasks in general purpose OS.

SMART combines the best-effort real-time scheduling into the fair queuing and achieves so far the most attractive properties for supporting multimedia and real-time applications, however, it relies on future workload prediction, thus, the scheduling results are sensitive to the prediction errors. Also, SMART suffers from its combinational high complexity that prevents its practical use. BVT is an effective yet low-overhead algorithm in supporting time-sensitive tasks in general purpose OS, it employs a warping mechanism to give scheduling preference to time-sensitive tasks and achieve their time-constraint meeting. However, more researches on the warping mechanism are required to deal with the interaction of multiple warped tasks and achieve a proper setting of warp parameters.

3 Energy-based fair queuing scheduling

In this chapter, we present the energy-based fair queuing scheduling to properly distribute energy among different tasks; by combining it with an epoch mechanism that throttles the energy dissipation to restrict the battery discharge rate, we propose a power management scheme that guarantees a user-specified battery lifetime to a target application in OS-based mobile systems.

First, the concept of epoch is introduced to achieve a target lifetime by throttling the energy dissipation in each epoch. Then, under certain assumptions on the system energy consumption, the energy-based scheduling model is built based on its similarity to the models of packet-based fair queuing in network scheduling and time-quantum-based fair queuing in CPU scheduling. Next, based on the requirements of the energy-based fair queuing and a comparative analysis of the properties of existing fair queuing algorithms, the starting-energy fair queuing (SEFQ) is proposed on the basis of the well-known starting-time fair queuing. After that, the features of the power share of periodical tasks are analyzed, and mechanisms are proposed to protect and reallocate the power share of time-sensitive tasks. Later on, the different methods to meet time constraints under energy-based fair queuing are discussed. Finally, by combining a real-time friendly mechanism into the SEFQ, the borrowed starting-energy fair queuing is proposed to better support time-sensitive tasks.

3.1. Throttling the energy dissipation

Let us consider a mobile operating system with a set of tasks $\{T_1, \dots, T_n\}$ (periodical real-time task, interactive task, and batch task2) competing for the total amount of energy E_{total} available in the battery. The system has a target lifetime T_{target}, which is divided into m number of periods of time termed epoch, with length T^i_{epoch}. In each epoch, the energy available for consumption, known as E^i_{epoch}, is limited so that if E^i_{epoch} is exhausted before the end of an epoch, the CPU is forced to be idle even if there are tasks still waiting to be executed. On the other hand, if E^i_{epoch} is not exhausted during the current epoch, the remaining amount of energy will be reclaimed by the system. How this reclaimed energy should be reused to maximize the application performance without harming the goal of target lifetime has been researched in [3] and will be further explored in our next-stage work. Mathematically, the average power P^i_{epoch} over one epoch is,

$$P^i_{epoch} = \begin{cases} \frac{E^i_{epoch}}{T^i_{epoch}}, & E^i_{epoch} \text{ exhausted in } T^i_{epoch} \\ \frac{E^i_{epoch-}}{T^i_{epoch}}, & E^i_{epoch} \text{ not exhausted in } T^i_{epoch} \end{cases} \quad (3.\ 1)$$

where E^i_{epoch-} denotes the actual energy consumed within one epoch. The average system power within the whole target lifetime can be computed as,

$$P_{target} = \frac{\sum_{i=1}^{m} P^i_{epoch} \cdot T^i_{epoch_i}}{T_{target}}, \sum_{i=1}^{m} P^i_{epoch} \cdot T^i_{epoch_i} \leq E_{total} \quad (3.\ 2)$$

By dividing the operational time into epochs and limiting the available energy within each epoch, the average system power and thus the battery discharge rate during the target lifetime are restricted to ensure the achievement of the target lifetime. Moreover, if the system activity is throttled in a proper way that establishes continuous energy consumption of modest level, the capacity of battery can also be extended [68], [69] and [70]. The total energy E_{total} can be flexibly allocated to variable-length epochs with variable size of E^i_{epoch}, thus supports different levels of performance in different stages of the lifetime. This is a useful property, considering again the scenario of a live

[2] In this thesis, batch task also refers to non-real-time task or regular task

football match broadcast, in which users may expect high video quantity in the beginning of the match and along the time a degrading quantity in accordance with the remaining energy in the battery.

3.2. The Energy-based scheduling model

For simplicity, there assumptions are made in the current stage of our work:

- First, the system base power consumption without running any task is assumed to be zero. This assumption does not affect the rationality of our model, since the basic system power can be simply combined into the model by adding a constant value.

- Second, all energy consumption of a task is attributed to its execution on the CPU. This is not true in a real mobile system where energy can also be consumed in other components such as memory, data and address buses. However, due to the fact that energy is managed as the first-class resource in our scheme, it can be easily extended to manage energy in a system-wide manner by taking into account the energy consumption of other components of the system.

- Third, since this work is focused on the scheduling algorithm, we assume that energy consumption in devices can be properly mapped to specific tasks.

With the concept of epoch introduced, the scheduling problem can be concentrated on one epoch by considering the same set of tasks $\{T_1, \ldots, T_n\}$ (periodical real-time task, interactive task, and batch task) competing for the limited amount of energy E^i_{epoch} during the i^{th} epoch. Similar to the network bandwidth and CPU bandwidth, energy is a limited resource commonly shared by different tasks with various energy requirements. Equivalence can be built between the (network or CPU) bandwidth allocation and energy allocation. In packet-based network scheduling, the service received by each session is data measured in bits, each session is allocated a share of network bandwidth that is defined as its rate of data transmission and measured in bits per seconds (bps); in time-quantum-based CPU scheduling, the

service received by each task is time measured in CPU clock cycles3, each task is allocated a share of CPU bandwidth that is defined as the rate of its execution on CPU and measured in cycles per second or in Hz; similarly, we can have an energy-based fair queuing scheduling if the energy measured in Joules is considered as the service, then each task will be allocated a share of power that is defined as its rate of energy consumption and measured in Joules per second or Watt. Therefore, the GPS model can also be applied to model the energy sharing among different tasks.

Ideally, we assume a GPS scheduler can simultaneously and proportionally serve energy to multiple tasks through the CPU. A task can be in two states: active if it is competing for the energy and passive if it is not. Each task T_i is assigned a weight w_i, which determines the minimum power sharing of the task. Then, in reference to equation (2. 2) and (2. 16), each task is executed at least with a power that equals to,

$$P_i(t) = P(t) \cdot \frac{w_i}{\sum_{\forall j \epsilon A(t)} w_j} \tag{3. 3}$$

where $P(t)$ denotes the variable CPU power along the time, and $A(t)$ denotes the set of active tasks at time t. For any interval $(t_1, t_2]$ during which the set of active tasks does not change, the energy consumed by task T_i is,

$$E_i(t_1, t_2) = w_i \int_{t_1}^{t_2} \frac{P(\tau)}{\sum_{j \in A(\tau)} w_j} d\tau \tag{3. 4}$$

In the real implementation, energy is consumed by allowing tasks running on the CPU for a certain period of time, and, at one time only one task can be executed. In our energy-based model, the basic unit for time quantum is one CPU time unit (Tu), which if necessary, can be a fixed number of CPU clock cycles, and the basic unit for measuring energy consumption is one energy unit (Eu), thus the basic unit for measuring power is Eu/Tu, defined as one power unit (Pu). As in the time-sharing CPU scheduling, CPU is allocated to tasks in the form of discrete time quantum of maximum Q Tus, thus, the energy is allocated to tasks along with the discrete CPU time quanta. A CPU time quantum with a length of Q Tus is called the standard time quantum. For task T_i whose service time is divided into m time quanta $\{q_i^j\}, j = 1,2, \dots, m$, the amount of energy consumption during its k^{th} time quantum q_i^k is defined as an energy packet, represented by e_i^k and measured in Eus. The size of an energy packet depends on the

[3] We will use cycle or CPU cycle to represent CPU clock cycle in the remaining of the paper

power function of the executed task as well as the length of its corresponding time quantum, thus, it varies in each energy packet. In this paper, time quantum q_i^k and energy packet e_i^k are collectively called as service quantum es_i^k.

A detailed comparison among the scheduling models of network, CPU and energy is listed in Table 3. 1.

	Network	**CPU**	**Energy**
Scheduling Resource	Data in bits $W_i(t_1, t_2)$	CPU in cycles $S_i(t_1, t_2)$	Energy in Eus $E_i(t_1, t_2)$
Allocation /Share of	Bandwidth in bps $C_i(t)$	Bandwidth in Hz $f_i(t)$	Power in Pus $P_i(t)$
Scheduling Objects	Sessions	Tasks	Tasks
Minimum Schedulable Unit	Packet in bits	Time quantum in cycles	Time quantum in Tus /Energy packet in Eus

Table 3. 1 Relationship among the scheduling models of network, CPU and energy

3.3. Starting-energy fair queuing

Based on the energy model, a fair queuing scheduling algorithm schedules tasks according to their received energy. However, many issues may arise if we directly apply an existing fair queuing scheduling algorithm to the energy domain. First, unlike the packet size that can be known upon arrival in network scheduling, the energy consumption in the future is not known in advance and hard to predict considering the fact that the power varies among tasks, and within the same task the power varies depending on which piece of code is being executed, especially in the case of multimedia applications. Even if energy prediction mechanisms are available, how the prediction accuracy may affect the performance of these algorithms is another open issue [38]. Therefore, packet-based fair queuing algorithms such as WFQ, WF^2Q, SCFQ and SPFQ are not suitable for our energy model. Time-quantum-based CPU fair queuing algorithms like the stride scheduling and EEVDF provide a mechanism to deal with the unknown length of service quantum, thus also address the unknown size of energy packet issue. However, extra overhead is introduced because the finishing tag has to be modified to reflect the actual size of energy packet at the end of its execution. Besides, the CPU power varies along the time depending on which task is executed on the CPU at that moment, while in most fair queuing algorithms the network or CPU bandwidth is assumed to be constant. Those time-quantum-based algorithms that

update the system virtual time by assuming a constant CPU bandwidth, such as stride scheduling, EEVDF and SMART, may fail to provide their claimed fairness and delay bound under variable CPU power. Last but not least, a practical energy-based scheduling algorithm should avoid introducing high time complexity, thus, fair queuing algorithms that require the ideal fluid-flow system to be run simultaneously are not good candidates for energy-based scheduling.

According to the above requirements as well as the discussion and comparison of existing fair queuing algorithms in section 2.3, we consider the Starting-time fair queuing (SFQ) as the better candidate to be applied in energy-based scheduling. In this section, we propose the Starting-energy fair queuing (SEFQ), a variation of SFQ based on the energy-based model introduced in section 3.2.

SEFQ defines the starting tag as the normalized energy received by a task, and schedules tasks in the increasing order of the starting tag. To compute the starting tag, a time function named virtual energy is defined to keep track of both the received and missed normalized energy of each task, similar to the concept of virtual time in traditional fair queuing algorithms. This is realized by defining a task virtual energy $V_i(t)$ to keep track of the normalized energy received by each task, and a system virtual energy $V(t)$ to keep track of the normalized energy consumed in the system. The system virtual energy works as a reference to update the value of the task virtual energy whenever a task leaves the system temporarily and later rejoins or a new task joins the system. The starting tag is defined as the value of the task virtual energy $V_i(t)$ at the instant of time when the k^{th} service quantum q_i^k of task i begins execution. Let $AR(q_i^k)$ denotes the time at which the service quantum q_i^k is requested. If the service quantum q_i^k is requested at the moment that task i is making a transition from passive state to active state, then $AR(q_i^k)$ equals to the time at which the transition is made; otherwise, $AR(q_i^k)$ equals to the moment when the previous service quantum q_i^{k-1} of task i finishes execution. Then, the starting tag S_i^k of service quantum q_i^k is computed as,

$$S_i^k = \max\{V(AR(q_i^k)), F_i^{k-1}\} \quad (3.5)$$

where $V(t)$ is the system virtual energy defined to be equal to the starting tag of the task in-service at time t, and F_i^{k-1} is the finishing tag of the previous service quantum $q_i^{k-1}$of task i, defined as the value of the task virtual energy $V_i(t)$ at the instant of time when q_i^{k-1} finishes execution. F_i^k is incremented as,

$$F_i^k = S_i^k + \frac{e_i^k}{w_{i,}}, F_i^0 = 0 \quad (3.6)$$

Since the starting tag of the task currently in-service is the minimum starting tag of all active tasks, the system virtual energy $V(t)$ in SEFQ is non-decreasing function that tracks the lowest virtual energy of all active tasks.

SEFQ inherits the following properties from SFQ [34] and [37]:

1. SEFQ is computationally efficient with time complexity of O(1) for starting tags computation and O(log N) for service quantum selection. Since in SEFQ, the system virtual energy is computed in reference to the real implementation itself, there is no need to simultaneously run the ideal GPS system for continuously recalculating the system virtual energy, which will lead to a notable computational overhead.

2. SEFQ does not require a priori knowledge of the service quantum length and energy packet size. Since SEFQ schedules tasks in the increasing order of starting tags, the length of the service quanta as well as the size of energy packets are not needed to be known in advance. This property is highly appreciated in multimedia task scheduling, where the service time as well as energy demand may vary dramatically and hard to predict.

3. SEFQ achieves a fair allocation of energy among tasks under variable CPU power. As shown in section 2.3.2.3.4, SFQ is able to provide fairness bound regardless of the fluctuations in system processor bandwidth. Since SEFQ has the same scheduling policy as SFQ, a fairness bound is also provided regardless of the variation of CPU power. For any interval $(t_1, t_2]$ during which two tasks m and n are continuously active in the system, the differences in the energy received by two tasks is:

$$\left|\frac{E_n(t_1,t_2)}{w_n} - \frac{E_m(t_1,t_2)}{w_m}\right| \leq \frac{e_n^{max}}{w_n} + \frac{e_m^{max}}{w_m} \quad (3.7)$$

where e_i^{max} denotes the maximum size of energy packet of task i.

4. SEFQ provides bounded delay and power guarantee to tasks under variable CPU power. As shown in section 2.3.2.3.4, SFQ is able to

provide bounded delay and throughput guarantee under a processing bandwidth that can be modeled as a fluctuation constrained (FC) Server. Intuitively, the long-term CPU power under fair queuing is fluctuated but constrained to an average power, with its top points bounded by the maximum task power and its bottom points bounded by the minimum task power. Thus, the variable CPU power function can be modeled as both burstiness constrained (BC) [67] that is upper-bounded and fluctuation constrained (FC) that is lower-bounded. Its upper bound burstiness and lower bound burstiness depends on the values of maximum and minimum task power, power shares of different tasks, and standard service quantum length Q. Since the fluctuation of CPU power under fair queuing is more strictly constrained than the FC model, SEFQ also achieves the bounded delay and throughput guarantee as achieved by SFQ.

5. Similar to SFQ, SEFQ provides lower delay to applications with low power share. Therefore, interactive applications of low energy requirements can have better response time.

The above properties inherited from SFQ are especially valued in the energy-based scheduling domain, in which the energy consumption is hard to predict and the CPU power varies along the time. A SEFQ scheduler allows tasks to proportionally consume energy according to their guaranteed power shares. Both excessive energy consumption and energy starvation can be avoided by allocating a proper power share, while unused or released power share can be proportionally allocated to the other active tasks in the system. Since the energy-centric scheduling proposed by Zeng and Ellis [1] and [3] is modified from the stride scheduling, neither detailed design nor theoretical foundation is provided, we believe SEFQ is the first clearly designed and formulated algorithm that provides all the above properties at the same time.

3.4. Insights into the power share

3.4.1 Maximum long-term power share and worst-case power share

Batch tasks can continuously receive energy until their total energy demands are met, and then finish their work and become completely passive. Therefore,

increase their weights will always increase their power shares. However, this is not true for periodical time-sensitive tasks[4]. Once a time-sensitive task receives its demanded energy in a period, it stops receiving energy and becomes passive until next period begins. Thus, different from batch tasks, increase the weight of a time-sensitive task may not always increase its power share. Further, in a long-term[5] static system where the set of active tasks, their weights and average powers do not change, the maximum long-term power share[6] that can be allocated to a time-sensitive task is bounded, and the bound depends not only on the weights of all active task, but also on the value of its average power[7] relative to the values of average powers of other active tasks in the system. Increase the weight of a time-sensitive task will increase its long-term power share to a value that is no greater than this maximum long-term power share, and the over-reserved share will be reallocated to the other active tasks in the system.

To demonstrate the idea of maximum long-term power share of periodical time-sensitive tasks, Table 3. 2 simply shows two tasks competing for the energy. For simplicity, we assume the length of each time quantum equals to the length of standard time quantum Q, and $\mathrm{Q} = 1\ \mathrm{Tu}$. R1 is a periodical real-time task and B1 is batch task. The energy packet size of B1 varies from 4 Eus to 8 Eus, with an average value of 6 Eus. R1 has a fixed-length period of 10 Tus, the service time in each period varies from 2 Tus to 6 Tus but averagely is 4 Tus, and the size of energy packet varies from 6 Eus to 10 Eus with its average size to be 8 Eus. In the long-term, the average amount of energy consumed by R1 in one period is at most $4 \times 8 = 32\ \mathrm{Eus}$, even if R1 is assigned a infinitely high weight to allow it consume energy without limit in each period, thus the maximum long-term power of R1 is $\frac{4\times8}{10} = 3.2\ \mathrm{Pus}$; since averagely a service time of at

[4] In this paper, periodical real-time task and interactive task are collectively called periodical time-sensitive task, interactive task can be seen as a periodical task with unfixed length of period.

[5] The long-term here is statistical, for the average power of a periodical time-sensitive task, it is relative to the period; and for the average power of a batch task, it is relative to a standard time quantum.

[6] The maximum long-term power share is the maximum statistical/average power share that can be allocated to a periodical task in the long-term. It guarantees in the long-term all energy demands of a periodical task will be met.

[7] The average power is assumed to be constant during an interval that can be seen as long-term.

least 6 Tus is left to B1 in each R1 period in the long term, B1 will consume at least $6 \times 6 = 36\,\mathrm{Eus}$ in each period, and its minimum long-term power is $\frac{6\times6}{10} = 3.6\,\mathrm{Pus}$. Therefore, the maximum long-term power share of R1 is $\frac{3.2}{3.2+3.6} = 0.47$, and the minimum long-term power share of B1 is 0.53. On the other side, the minimum long-term power share of R1 can infinitely approach zero if R1 is assigned an infinitely low weight, and the maximum long-term power share of B1 can infinitely approach one if B1 is allocated an infinitely high weight.

The maximum long-term power share guarantees meeting all the energy demands of a periodical task in the long-term, thus, allocate time-sensitive tasks a weight that guarantees a maximum long-term power share is a prerequisite for meeting majority of the time constraints. In other words, if a time-sensitive task has a power share less than its maximum long-term power share, it will miss most of its time constraints. Since there is no sense to keep executing a real-time task on a system if it misses most of its time-constraints, periodical time-sensitive tasks should be assigned a weight to guarantee a power share that at least equals to the maximum long-term power share. The weight corresponding to the maximum long-term power share is called the minimum weight $\mathrm{w}_{\mathrm{min}}^{\mathrm{i}}$ for meeting time-constraints.

A power share guarantees meeting all the energy demands in the long-term does not necessarily guarantee meeting all the energy demands in each period. To meet all the energy demand in each period, a power share corresponding to the worst-case execution time (WCET) and maximum size of energy packet is required. In this paper, the amount of energy demanded in each period is called energy load. When a periodical real-time task has the worst-case execution time and maximum size of energy packet at the same time, it has the worst-case energy load. The power share corresponding to the worst-case energy load is called the worst-case power share, and the weight corresponding to the worst-case power share is called the worst-case weight $\mathrm{w}_{\mathrm{wc}}^{\mathrm{i}}$. The worst-case power share depends not only on the worst-case energy load of the concerned task, but also on the energy load of other tasks. Theoretically, the worst-case power share is computed when the concerned task has the worst-case energy load and all the other tasks have the lowest energy load. However, in real system the probability of the above situation is very small, therefore a practical worst-case power share can be used considering the looser situation when the concerned task has the worst-case energy load and all the other tasks have their long-term average energy load.

Consider again the tasks in Table 3. 2, the maximum amount of energy consumed by R1 in one period is $6 \times 10 = 60\ \mathrm{Eus}$, thus the maximum power of R1 in one period is $\frac{6\times10}{10} = 6\ \mathrm{Pus}$; B1 uses the left 4 Tus, consumes at least $4 \times 4 = 16\ \mathrm{Eus}$, with a power of $\frac{4\times4}{10} = 1.6\ \mathrm{Pus}$. Therefore, without considering the allocation error, theoretically a worst-case power share of $\frac{6}{6+1.6} = 0.79$ is required to meet the energy demands of R1 under its worst-case energy load. To compute the practical worst-case power share, the power of B1 is computed considering its average size of energy packet, so in the left 4 Tus, B1 has a average power of $\frac{6\times4}{10} = 2.4\ \mathrm{Pus}$. Thus, the practical worst-case power share of R1 is $\frac{6}{6+2.4} = 0.714$. Since the probability for R1 to have the maximum power in each period is $\frac{1}{5} \times (\frac{1}{5})^6 = \frac{1}{78125}$, in real system this power share is enough to meet all the energy demands of each period.

Task	R1	B1
Period (Tu, standard time quantum $\mathrm{Q} = 1\ \mathrm{Tu}$)	10	N/A
Number of service quanta / period	2-6	N/A
Average number of service quantum / period	4	N/A
Size of energy packet (Eu)	6-10	4-8
Average size of energy packet (Eu)	8	6
Maximum long-term power share	0.47	$\mathrm{x} \to 1$
Minimum long-term power share	$\mathrm{x} \to 0$	0.53
Theoretical worst-case power share	0.79	N/A
Practical worst-case power share	0.714	N/A

Table 3. 2 Maximum long-term and worst-case power share computation with 2 tasks

The maximum long-term power share and worst-case power share only keep constant in a long-term static system where the set of active tasks, their weights and average powers do not change. In a real dynamic system, tasks may frequently join or leave the system, change their weights or vary their average powers[8] (or average energy loads) in different long-term intervals. Therefore, both the maximum long-term power share and the worst-case power share are variable.

[8] For simplicity, in this thesis we assume that during each epoch the average power of each task is statistically constant.

Table 3. 3 shows the recalculation of maximum long-term power share and worst-case power share when the periodical real-time task R2 and batch task B2 join to the system of Table 3. 2. The two real-time tasks R1 and R2 are assigned at least their minimum weights. Batch tasks B1 and B2 are assigned a weight of 1 and 2, respectively, thus the average size of energy packet of the two batch tasks is $\frac{6\times1+12\times2}{3} = 10\ \mathrm{Eu}$. In the long term, the average power of R1 is $\frac{4\times8}{10} = 3.2\ \mathrm{Pus}$, the average power of R2 is $\frac{3\times15}{15} = 3\ \mathrm{Pus}$, and two batch tasks together have an average power of $\left(1-\frac{4}{10}-\frac{3}{15}\right)\times 10 = 4\ \mathrm{Pus}$. Therefore, the maximum long-term power share of R1 and R2 is 0.341 and 0.294, respectively. Since the worst-case power of R1 is $\frac{6\times10}{10} = 6\ \mathrm{Pus}$, the lowest power of R2 is $\frac{2\times12}{15} = 1.6\ \mathrm{Pus}$, and the lowest power of two batch tasks together is $\left(1-\frac{6}{10}-\frac{2}{15}\right)\times\frac{4\times1+10\times2}{3} = 2.16\ \mathrm{Pus}$, theoretically the worst-case power share of R1 is 0.615. Considering the average power of R2 being $\frac{3\times15}{15} = 3\ \mathrm{Pus}$, and the average power of two batch tasks being $\left(1-\frac{6}{10}-\frac{3}{15}\right)\times 10 = 2\ \mathrm{Pus}$, the practical worst-case power share is 0.545. Similarly, the theoretical and practical worst-case power share of R2 is 0.468 and 0.424, respectively.

Task	R1	R2	B1	B2
Period (Tu, standard time quantum $\mathrm{Q} = 1\ \mathrm{Tu}$)	10	15	N/A	N/A
Number of service quantum / period	2-6	2-4	N/A	N/A
Average number of service quantum / period	4	3	N/A	N/A
Size of energy packet (Eu)	6-10	12-18	4-8	10-14
Weight	$\geq \mathrm{w}_{\mathrm{min}}^{\mathrm{R1}}$	$\geq \mathrm{w}_{\mathrm{min}}^{\mathrm{R2}}$	1	2
Average size of energy packet (Eu)	8	15	6	12
			10	
Maximum long-term power share	0.314	0.294	$\mathrm{x}\to 1$	$\mathrm{x}\to 1$
Theoretical worst-case power share	0.615	0.468	N/A	
Practical worst-case power share	0.545	0.424	N/A	

Table 3. 3 Maximum long-term and worst-case power share computation with 4 tasks

3.4.2 Power share protection

To meet time constraints, real-time tasks should be allocated a desired power share that is not affected by dynamic task participations. The value of the desired

power share can be the worst-case power share if it is necessary to meet all the deadlines, or between the maximum long-term power share and the worst-case power share if some number of deadline missed is acceptable. However, from equation (3. 3), we can see the power share P_i of a task varies with the number and weights of active tasks in the system. Any new task joins the competition with a large weight may significantly reduce the power share of a time-sensitive task to a value that is lower than its maximum long-term power share, leading to the miss of the majority of its deadlines. Thus, a share protection mechanism is required to guarantee a specified power share to time-sensitive tasks.

Our solution to achieve power share protection is similar to the one proposed by Al-Ouran [46]. In order to flexibly support a mix of applications with wider categories, each task is assigned an initial power share $\overline{P_i} \in [0,1]$ and an initial weight $\overline{w_i}$. As in section 2.3.2.2, tasks are classified into three categories:

- $\overline{P_i} = 0$ and $\overline{w_i} > 0$, if the task is a regular task does not require a guaranteed power share. This task competes for unreserved CPU power with other tasks based on its initial weight, e.g. a gcc compile, or a Linux grep.

- $\overline{P_i} > 0$ and $\overline{w_i} = 0$, if the task is a time-sensitive task that only asks for a guaranteed share. The zero weight means the task does not compete for unreserved or any released CPU power, e.g. a guest operating system running as a user-level task, or any hard real-time task with constant energy load per period.

- $\overline{P_i} > 0$ and $\overline{w_i} > 0$, if the task is a time-sensitive task that not only requires a guaranteed power share but also competes for unreserved or any released CPU power. The task competes for energy with other tasks based on its initial weight, e.g. multimedia player.

The effective power shares and weights are computed based on the initial power shares and weights. We only consider the under-load situation, in which the sum of initial power shares of all active tasks is less than 100% ($\sum_{j \in A(t)} \overline{P_j} < 1$). The effective power share is computed by the equation (2. 30), with the bandwidth share f_i replaced by power share P_i,

$$P_i = \overline{P_i} + \left(\frac{\overline{w_i}}{\sum_{j \in A(t)} \overline{w_j}}\right) \cdot (1 - PS),\ PS = \sum_{j \in A(t)} \overline{P_j} \qquad (3.\ 8)$$

Where PS denotes the total power share reserved by active time-sensitive tasks. However, we calculate the effective weight w_i in an easier way. Instead of referring equation (2. 31) to compute the effective weight, we equal the effective weight to the effective power share by $w_i = P_i$. The power shares allocated to tasks are the same as in [46], but the recalculation of effective weights is significantly simplified. Consequently, this share protection mechanism is suitable for systems with arbitrary combination of regular tasks and time-sensitive tasks.

To demonstrate the idea of the share protection mechanism, Table 3. 4 shows the recalculation of the effective share P_i and effective weight w_i when new tasks join to the resource competition.

Task	$\bar{P}_i$	$\bar{w}_i$	P_i	w_i
1	0.2	0	0.2	0.2
2	0.2	1	0.3	0.3
3	0	2	0.2	0.2
4	0	3	0.3	0.3

(a)

Task	$\bar{P}_i$	$\bar{w}_i$	P_i	w_i
1	0.2	0	0.2	0.2
2	0.2	1	0.25	0.25
3	0	2	0.1	0.1
4	0	3	0.15	0.15
➢ **5**	0	6	0.3	0.3

(b)

Task	$\bar{P}_i$	$\bar{w}_i$	P_i	w_i
1	0.2	0	0.2	0.2
2	0.2	1	0.225	0.225
3	0	2	0.05	0.05
4	0	3	0.075	0.075
5	0	6	0.15	0.15
➢ **6**	0.3	0	0.3	0.3

(c)

Table 3. 4 Recalculation of effective share and effective weight

In Table 3. 4(a), there are four tasks active in the system. Task 1 and 2 are time-sensitive tasks that require a guarantee of power share. Especially, task 2 can compete for unreserved or any released CPU power share with its initial weight. Task 3 and 4 are regular tasks which do not require a guaranteed share. Task 1 and 2 reserve a total power share of $= 0.4$, then the remaining power share $1 - PS = 0.6$ is allocated to task 2, 3, and 4 with their initial weights being 1, 2, and 3, respectively. The effective shares are computed using equation (3. 8):

$$P_1 = 0.2 + 0 = 0.2,$$

$$P_2 = 0.2 + \frac{1}{6} \times 0.6 = 0.3,$$

$$P_3 = 0 + \frac{2}{6} \times 0.6 = 0.2,$$

$$P_4 = 0 + \frac{3}{6} \times 0.6 = 0.3.$$

The effective weight equals to the effective share.

Table 3. 4(b) shows the recalculation of effective values when regular task 5 joins to the energy competition with initial share $\overline{P}_5 = 0$ and initial weight $\overline{w}_5 = 6$. Using equation (3.5), the new effective shares are:

$$P_1 = 0.2 + 0 = 0.2,$$

$$P_2 = 0.2 + \frac{1}{12} \times 0.6 = 0.25,$$

$$P_3 = 0 + \frac{2}{12} \times 0.6 = 0.1,$$

$$P_4 = 0 + \frac{3}{12} \times 0.6 = 0.15,$$

$$P_5 = 0 + \frac{6}{12} \times 0.6 = 0.3.$$

Task 5 only compete for the remaining 0.6 power share, the reserved power shares of task 1 and 2 are not affected.

Table 3. 4(c) shows the recalculation of effective values when time-sensitive task 6 joins to the energy competition with initial share $\overline{P}_6 = 0.3$ and initial weight $\overline{w}_6 = 0$. In this case, the total power share reserved for time-sensitive tasks is $PS = 0.7$, the effective shares are recalculated as，

$$P_1 = 0.2 + 0 = 0.2,$$

$$P_2 = 0.2 + \frac{1}{12} \times 0.3 = 0.225,$$

$$P_3 = 0 + \frac{2}{12} \times 0.3 = 0.05,$$

$$P_4 = 0 + \frac{3}{12} \times 0.3 = 0.075,$$

$$P_5 = 0 + \frac{6}{12} \times 0.3 = 0.15,$$

$$P_6 = 0.3 + 0 = 0.3.$$

The reserved power share $PS = 0.7$ is first guaranteed to time-sensitive tasks 1, 2, and 6, then the remaining 0.3 power share is allocated to tasks 2, 3, 4, and 5 proportionally to their initial weights.

3.4.3 Power share reallocation

When a task finishes its work and leaves the competition, its share of CPU power should be released and reallocated to the other active tasks in the system. In an ideal model, the released power share is fairly reallocated to other active tasks according to their initial weights. However, unfair share reallocation will occur in a real system if the power shares of periodical time-sensitive tasks are released and reallocated when they finish their work in one period and temporarily leaves the system until the beginning of the next period. This is because when there are multiple periodical time-sensitive tasks in the system, depends on which time-sensitive task or which set of time-sensitive tasks temporarily release the power share, the remaining power share available for competition varies from time to time. This problem has not been dealt with in former share protection mechanisms.

To demonstrate how the unfair reallocation may occur in a real system, we consider a set of tasks listed in Table 3. 5.

Task	$\bar{P}_i$	$\bar{w}_i$	P_i	w_i
1	0.1	0	0.1	0.1
2	0.2	0	0.2	0.2
3	0.4	0	0.4	0.4
4	0	1	0.1	0.1
5	0	2	0.2	0.2

Table 3. 5 Reference tasks for showing unfair power share reallocation

In Table 3. 5, tasks 1, 2 and 3 are periodical time-sensitive tasks with non-zero initial shares while tasks 4 and 5 are regular tasks with non-zero initial weights. The real-time class tasks reserve a total share of 0.7, and the remaining 0.3 power share is allocated to task 4 and 5 proportionally to their initial weights. Therefore, when all the five tasks are active in the system, the effective weights of task 4 and 5 are 0.1 and 0.2, respectively. Assume that at a certain moment, task 1 finishes its work of one period, temporarily leaves the system and releases its power share of 0.1. If task 4 is the next

task to be executed, it will advance the virtual energy with an effective weight of $\frac{(1-0.6)\times 1}{3} = 0.13$. At another moment, task 5 is executed after task 1 and 2 both leave the system and release their total share of 0.3, then task 5 will advance the virtual energy with an effective weight of $\frac{(1-0.4)\times 2}{3} = 0.4$. Since task 4 and 5 no longer advance their virtual energy in accordance with their initial weights 1 and 2, they fail to receive energy fairly and proportionally in the long-term.

One solution to the above problem is to fix the total power share of periodical time-sensitive tasks PS when any of them temporarily leaves the system because of finishing the work in one period. However, PS will be changed if they finish all the periods of work and completely leave the system. This solution, although favors other active time-sensitive tasks by allocating them a larger power share, supports fair and proportional energy allocation among these tasks with non-zero initial weights. With this solution introduced, the way to compute the effective weights and shares changes slightly, because the sum of all effective power shares is always one while the sum of all effective weights is not required to be one. When any time-sensitive task temporarily leaves the system, the effective weight is directly re-computed from the initial power shares of all time-sensitive tasks and initial weights of all active tasks,

$$\mathrm{w_i} = \overline{\mathrm{P_i}} + \left(\frac{\overline{\mathrm{w}}_\mathrm{i}}{\sum_{\mathrm{j}\epsilon\mathrm{A(t)}} \overline{\mathrm{w}}_\mathrm{j}}\right) \cdot (1 - \mathrm{PS}) \tag{3. 9}$$

where PS is the sum of initial power shares reserved by all time-sensitive tasks (including active and temporarily passive ones). Since the sum of effective weights is not one anymore, the effective power share is re-computed as,

$$\mathrm{P_i} = \frac{\mathrm{w_i}}{\sum_{\forall \mathrm{j}\epsilon\mathrm{A(t)}} \mathrm{w_j}} \tag{3. 10}$$

To demonstrates the idea of the above solution, Table 3. 6 shows the recalculation of power shares when real-time tasks of Table 3. 5 leave the system temporarily. Table 3. 6(a) shows the power share reallocation when task 3 leaves the system. The total reserved power share PS is fixed as 0.3, according to equation (3. 9) the weights of all active tasks keep the same as the original ones. However, since the total weights of all active tasks is no longer one, the effective power shares change proportionally to the effective weights and are re-computed according to equation (3. 10). A larger power share is allocated to task 1 and 2 so that they can be scheduled more frequently and meet more time constraints, and if the allocated power share is larger than the power share corresponding to the actual energy load, task 1 and 2 will

finish their work before the end of their period and release their power shares to the rest active tasks. Table 3. 6(b) shows the power share reallocation when task 2 leaves the system. As we can see, in both cases of task leaving, task 4 and 5 receive their power shares proportionally to their initial weights. In the long-term, if the maximum long-term power shares of task 1, 2 and 3 are P_{max}^{1}, P_{max}^{2} and P_{max}^{3} , respectively, then the long-term power shares of task 4 and 5 are $\frac{1}{3} \times (1 - \sum_{i=1}^{3} P_{max}^{i})$ and $\frac{2}{3} \times (1 - \sum_{i=1}^{3} P_{max}^{i})$, respectively. Therefore, this solution improves real-time performance while guarantees proportional share of energy among different tasks in a real system.

Task	$\bar{P}_i$	$\bar{w}_i$	P_i	w_i
1	0.1	0	0.167	0.1
2	0.2	0	0.333	0.2
4	0	1	0.167	0.1
5	0	2	0.333	0.2

(a)

Task	$\bar{P}_i$	$\bar{w}_i$	P_i	w_i
1	0.1	0	0.25	0.1
4	0	1	0.25	0.1
5	0	2	0.5	0.2

(b)

Table 3. 6 Power share reallocation

3.5. Time-constraint meeting

Meeting time constraints of a time-sensitive task is equivalent to meeting the energy demands in each period. Since the energy load of multimedia and most soft real-time applications usually varies in different periods, the power share required for meeting the energy demand in each period is also variable. In CPU fair queuing scheduling, adaptive share is proposed to deal with the issue of variable workload. However, there are several practical issues to apply adaptive power shares in the energy domain. First, the proper power share in each period should be recalculated based on the historical energy demands of previous periods. Since it relies on the energy consumption prediction, the real-time performance is sensitive to the prediction error. Also, the power share recalculation and energy prediction will bring overhead to the scheduling algorithms. Second, unlike the CPU scheduling in which the proper bandwidth share can be easily computed based on the predicted service time and the length of its period, the proper power share in each period is difficult to compute because it depends not only on the energy demands in one period but also on the power of other active tasks.

A simple method to meet time constraints is to assign time-sensitive tasks an initial power share that at least equals to its maximum long-term power share. However, it leads to an over-reservation of power share because in most cases the actual power share required by one time-sensitive task is lower than the reserved power share. The over-reserved power share is wasted in the sense that it cannot be reserved by other time-sensitive tasks. Furthermore, in a dynamic system where the worst-case power share may change frequently and significantly, the initial power shares reserved for time-sensitive tasks have to be adjusted timely and frequently, which brings challenge to time-constraint meeting and increases the overhead on the scheduler. Last but not least, although theoretically a worst-case power share is enough to guarantee meeting all the time constraints, in real implementation a higher power share is required considering the energy allocation error caused by the quantization. However, it is difficult to take the energy allocation error into account when computing the worst-case power share. The main difficulty comes from two aspects: first, similar to the allocation error (lag) in SFQ, the energy allocation error under SEFQ increases linearly to the number of active tasks due to its failure to meet the minimum slope property, according to equation (2. 35), it is impossible to compute the allocation error under variable CPU power without knowing the average CPU power P and the power burstiness $\delta(P)$; second, again the power share not only depends on the energy allocation error itself but also depends on the power of other active tasks. As a result, the exact minimum power share required to meet the worst-case energy load cannot be determined. It means a coarse, conservative power share that may be overly larger than the worst-case power share has to be reserved to one time-sensitive task. Moreover, the worst-case power share of one time-sensitive task may change significantly in a dynamic system; deadlines may be missed if the power share reserved for a time-sensitive task is not timely adjusted according to the latest worst-case power share.

In fair queuing scheduling, maintaining a strict fairness is a double-edged sword for supporting time-sensitive tasks. On one side, time constraints cannot be effectively met as a result of the scheduler taking away the time-sensitive task from the CPU at an inopportune time in trying to ensure the fairness; on the other side, achieving an optimal fairness bound and implicitly a minimum allocation error can minimize the share over-reservation for time-sensitive tasks, thus, save the share reservation space for supporting more time-sensitive tasks. In SEFQ, however, the near-optimal fairness bound does not guarantee a stable allocation error that is unaffected by the number of active tasks, and as mentioned above, the exact minimum power share required to meet all the time constraints cannot be determined, therefore, maintaining a strict

fairness becomes less attractive in supporting time-sensitive tasks. As in the CPU fair queuing scheduling, real-time friendly mechanisms can be combined into the SEFQ for better supporting time-sensitive tasks. A real-time friendly mechanism breaks the fairness between time-sensitive tasks and regular tasks by giving dispatch preference to time-sensitive tasks; however, it should still be able to constraint the long-term power share of time-sensitive tasks to avoid any energy starvation on regular tasks.

3.6. Borrowed starting-energy fair queuing

In this thesis, a real-time friendly mechanism named warping is combined into the SEFQ for better supporting time-sensitive tasks. The new algorithm is called borrowed starting-energy fair queuing (BSEFQ). The idea of warping in BSEFQ is similar to the one in the borrowed-virtual-time (BVT) scheduling [38], however, the scenario of multiple warped tasks is considered and their interaction is analyzed. Moreover, the way how the warp time limit works is simplified. Besides, some proposals are given to deal with the new issues that are brought into the energy domain by the warping mechanism.

3.6.1 The warping mechanism

BSEFQ records the normalized energy received by each task with the actual starting tag S_i, but schedules tasks in the increasing order of the effective starting tag. The effective starting tag ES_i is no greater than the original actual starting tag S_i. It can be equal to the S_i, defined as unwarped, or have a difference with S_i, defined as warped. The difference between the ES_i and S_i is called warp, represented by $warp_i$. Each task holds a parameter named $warpBack_i$ that decides whether the task can run warped ($warpBack_i = 1$) or unwarped ($warpBack_i = 0$). Regular tasks are not allowed to run warped so they are always assigned a $warpBack_i$ value of 0. Periodical time-sensitive tasks are assigned a $warpBack_i$ value of 1 at the beginning of each period, but the maximum time they can receive energy by running warped is limited by a parameter named warp time limit L_i. When the warp time limit L_i is reached, by changing the $warpBack_i$ value to be 0, one time-sensitive task is forced to become unwarped until the beginning of next period. Then, the effective starting tag ES_i is computed in a way that for time-sensitive tasks within the warp time limit it is the actual starting tag S_i minus the warp value, and in other cases it equals to the value of actual starting tag S_i. Let ES_i^k denotes the effective starting tag of the k^{th} service quantum es_i^k of task i, and $warp_i$ denotes the warp value allocated to task i, mathematically,

$$ES_i^k = \begin{cases} S_i^k - warp_i, & warpBack_i = 1 \\ S_i^k, & warpBack_i = 0 \end{cases} \quad (3.\ 11)$$

By warping back the starting tag and borrowing virtual energy from its future energy allocation, a time-sensitive task moves forward in the waiting queue and receives its share of energy earlier to strictly meet its time constraints [38]. The fairness between time-sensitive tasks and regular tasks is broken due to the dispatch preference given to warped time-sensitive tasks. However, the long-term power share of a time-sensitive task can still be restricted by the proportional fair sharing of SEFQ, because its actual starting tag S_i is advanced based on its actual energy consumption and effective weight and its maximum warp time is limited. In other words, if a time-sensitive task becomes unwarped before finishing all the service quanta, since its actual starting tag S_i has been advanced due to the energy consumption during the warped period, it will be placed at the very end of the waiting queue to let other batch tasks have the chance to catch up. By properly setting or adjusting the warp time limit of a time sensitive task, BSEFQ can trade off between its energy consumption controlling and time-constraint meeting. Therefore, on one side, the energy consumption of time-sensitive tasks of high energy load can be restricted to avoid draining the battery in a high rate and to protect the energy usage of other tasks; on the other side, time-constraints can be stringently met when energy is consumed in a normal rate.

When there are multiple warped tasks active in the system, warp values of different levels are assigned to tasks according to their priorities. The more important one task is, the higher warp value it has, correspondingly, the lower effective starting tag it has when it is warped. For example, a hard real-time task can be assigned the highest warp value, and a soft real-time task or an interactive task can be assigned a lower warp value. The task holding the highest warp value is dispatched immediately after its new period begins and can continuously occupy the CPU until finishing its work or reaching its warp time limit, while tasks holding lower warp value have to wait until more important tasks finish their work or become unwarped. Time-sensitive tasks of the same importance are assigned the same warp value, they are scheduled by turns, the dispatching frequency depends on the energy packet size and the effective weight, and their real-time performance can be traded off by changing the effective power shares. However, the minimum real-time performance of one time-sensitive task can be guaranteed by the warping mechanism with admission control.

By applying warp values of different levels, the warping mechanism combines the priority-based scheduling into the SEFQ. However, by setting the warp time limit, the maximum time one time-sensitive task can run with priority is limited. The priorities enable BSEFQ to flexibly and effectively support different types of time-sensitive tasks, while the warp time limit restricts the maximum power of time-sensitive tasks to avoid any energy starvation on regular tasks.

3.6.2 Proposals to deal with the system virtual energy

The warping mechanism brings a problem to the system virtual energy updating in SEFQ, which is not reported in BVT. Once a time-sensitive task is warped, it will be scheduled even if its actual starting tag S_i is larger than those of other batch tasks. Since the system virtual energy $V(t)$ in SEFQ is updated to be the S_i of the task currently in-service, it is no longer guaranteed to be less than or equal to the minimum virtual energy of all active tasks. In other words, $V(t)$ is no longer a non-decreasing function that always keeps pace with the lowest actual starting tag. If a new regular task joins the competition at time τ, since it is assigned an actual starting tag that equals to the $V(\tau)$, its execution will be delayed if $V(\tau)$ is larger than the lowest S_i in the system. The length of delay depends on the difference between $V(\tau)$ and the lowest S_i, which, unluckily can be considerable large under BSEFQ due to the different advancing speed of the virtual energy of time-sensitive tasks and regular tasks. When there are multiple time-sensitive tasks active the system, since they can be dispatched with a higher actual starting tag and the actual starting tag can be updated to the value of a larger system virtual energy (if that is the case, refer to equation (3.5)) when they transform from temporary passive state to active state, as time goes by, the virtual energy of time-sensitive tasks can get increasingly greater than those of the regular tasks. In this case, the actual starting tags of all time-sensitive tasks are catching with each other on a higher level, while the actual starting tags of all regular tasks are catching with each other on a lower level. Since only one $V(t)$ is available, it has to jump between the higher level virtual energy values and the lower level virtual energy values depending on the type of task executed. The difference between the higher level values and lower level values can be increasing greater without any restriction. As a result, a newly joined regular task will be significantly delayed in execution after the other regular tasks if its actual starting tag is updated to a higher level value of $V(t)$ at its joining moment; a newly joined time-sensitive task will be scheduled ahead of other

old time-sensitive tasks of the same priority if its actual starting tag is updated to a lower level value of $V(t)$ at its joining moment.

To solve the above problem, there are two options: modify the way system virtual energy is computed or modify the way the actual starting tag of newly joined tasks is updated.

In the first option, two functions of system virtual energy $V_{RT}(t)$ and $V_{Reg}(t)$ have to be employed to respectively track the virtual energy advancing of time-sensitive tasks and regular tasks. $V_{Reg}(t)$ is computed similarly to the computation of $V(t)$ in SEFQ. When a regular task is dispatched, $V_{Reg}(t)$ is updated to the value of the actual starting tag of the dispatched regular task; when a time-sensitive task is dispatched, no value update is needed for $V_{Reg}(t)$. Therefore, $V_{Reg}(t)$ is a non-decreasing function which always equals to the minimum actual starting tag of all regular tasks. The starting tag of any newly joined regular task is updated to the value of $V_{Reg}(t)$, thus, is able to be dispatched immediately. The computation of $V_{RT}(t)$ is a more complex issue due to the different priorities of time-sensitive tasks. Ideally, for the time-sensitive tasks of each priority, one $V_{RT}(t)$ should be available to track the virtual energy advancing. However, it is not practical to maintain one $V_{RT}(t)$ for each priority, because in a real mobile system it is less unlikely to have a number of time-sensitive tasks running in each priority. Using one $V_{RT}(t)$ to coarsely track the virtual energy advancing of all time-sensitive tasks is a compromising scheme, and there are several approaches to achieve that. A first approach is to update $V_{RT}(t)$ as $V_{Reg}(t)$, that is, $V_{RT}(t)$ is updated to the value of the actual starting tag of the currently dispatched time-sensitive task. In this case, $V_{RT}(t)$ is also not guaranteed to be a non-decreasing function, but different from the original system virtual energy $V(t)$, the maximum difference between its highest value and lowest value is restricted under certain value, which depends on the number and property of time-sensitive tasks and the parameters setting (e.g. initial power shares and weights, warp time limit) in the scheduler. Compared to the case of original $V(t)$, a newly joined time-sensitive task will be delayed slightly when competing with other time-sensitive tasks of the same priority if it joins at the moment a higher priority task is executed. A second approach is to guarantee a non-decreasing $V_{RT}(t)$, to achieve that, $V_{RT}(t)$ is updated to the value of the actual starting tag of the currently dispatched time-sensitive task only on condition that the value of the later one is larger than $V_{RT}(t)$, otherwise, $V_{RT}(t)$ keeps the same. In this case, $V_{RT}(t)$ is no less than the lowest actual starting tag among all real-time tasks, with their maximum difference restricted. Similar to the first approach, slight delay may be introduced to a newly joined

time-sensitive task. However, in the second approach, delay can happen to a newly joined task even if it joins the competition when a task of the same priority is executed.

In the second option, the key is to know the priority of the newly joined task and based on that, decide when its actual starting tag S_i should be updated. If a regular task T_i joins the competition when a time-sensitive task is being executed, instead of immediately updating S_i, T_i is still considered as idle until a regular task is being executed on the CPU, at which time S_i is updated to $V(t)$. Similarly, if a time-sensitive task T_i joins the competition when a time-sensitive task of higher priority is being executed, its S_i update is delayed to the time when a time-sensitive task of the same priority as T_i is executed. When the task being executed has higher priority than the newly joined task T_i, it is reasonable to delay the update of S_i because T_i cannot be dispatched immediately. However, if there is no old task that is of the same priority as the newly joined task T_i, S_i is immediately updated to $V(t)$. Also, when the task being executed has lower priority than the newly joined task T_i, S_i has to be updated to $V(t)$ immediately because T_i is dispatched immediately. Specifically, if a time-sensitive task T_i joins the completion when a regular task is being executed, S_i is updated to a lower level $V(t)$ value and T_i is dispatched immediately. However, if an old time-sensitive task T_j of the same priority as T_i joins the competition with a higher level S_j value after temporal idle, the execution of T_j will be delayed until all the service quanta of T_i are finished executing. To solve the problem, S_i has to be updated a second time, not to $V(t)$ but to S_j. In this case, T_i and T_j can compete fairly based on their effective weights. Since the actual starting tag S_i is strictly non-decreasing and the actual starting tag of a higher priority task is always higher than the one of a lower priority task, the update to S_j is only required once for a newly joined time-sensitive task.

As can be seen, the main difficulty is to maintain a system virtual energy that is able to track the virtual energy advancing of time-sensitive tasks. In this section, several potential approaches have been proposed to solve the updating problem of system virtual energy caused by the warping mechanism. The effectiveness of different approaches will be analyzed and compared based on SystemC simulation in our further work.

3.7. Summary

In this chapter, we have presented the energy-based fair queuing scheduling that manages energy as a first-class resource globally in the system and schedules

tasks according to the amount of energy received by each task. An energy-based fair queuing scheduling algorithm first proportionally serves energy to different tasks according to their assigned power shares, apart from that, effectively supports time-sensitive tasks in general purpose operating systems. By combining the energy-based fair queuing scheduling with an epoch mechanism that restricts the battery discharge rate, we can guarantee a user-specified battery lifetime to a target application in OS-based mobile systems.

Energy-based fair queuing is a cross-application of traditional fair queuing algorithms in the energy management domain, new design issues are incurred because the energy consumption in the future is not known in advance and the CPU power for sharing is variable along the time. SEFQ schedules tasks according to the virtual starting tag, which is able to be known before the dispatch decision is made; moreover, it updates the system virtual energy according to the virtual starting tags and applies an increasing slope of zero between any two update points, thus, the time-complexity for virtual starting tag computation is O(1) and no assumption of constant CPU power is made. SEFQ achieves proportional power sharing and provides a near-optimal fairness bound; however, the delay bound increases linearly to the number of active tasks in the system due to the increasing slope of zero between the system virtual energy updating.

Each periodical time-sensitive task has a maximum long-term power share that guarantees all its energy demands can be met in the long-term, and a worst-case power share that guarantees meeting all the energy demands in each period. To avoid missing most of the time constraints, a periodical time-sensitive task should be guaranteed a power share that is no lower than its maximum long-term power; therefore the idea of initial weights and effective weights is proposed to protect the power share of time-sensitive tasks against from dynamic energy competitions. Also, the total power share reserved for time-sensitive tasks is fixed to avoid the non-proportional virtual energy advancing caused by the randomly released power shares of time-sensitive tasks that temporally leave the competition.

To meet all the time constraints of a periodical time-sensitive task, a worst-case power share taking the allocation error into account should be allocated. However, it is difficult to compute the allocation error and combine it into the worst-case power share computation under a dynamic system, in which tasks may frequently join or leave the system, change their weights or vary their average powers. Therefore, a conservative power share that may be overly larger than the worst-case power share has to be

reserved to one time-sensitive task, which leads to a waste of share space. Moreover, the worst-case power share itself may change significantly in a dynamic system, if the power share reserved to a time-sensitive task is not adjusted timely, time constraints may be missed under SEFQ. In view of the conflict between meeting time-constraint and maintaining strict fairness, BSEFQ employs a real-time friendly mechanism named warping to breaks the fairness between time-sensitive tasks and regular tasks by giving dispatch preference to time-sensitive tasks. It combines the priority-based scheduling into SEFQ to flexibly and effectively supports different types of time-sensitive tasks; however, by limiting their maximum time of running in priority, the maximum power share of time-sensitive tasks can be constrained to avoid any energy starvation on regular tasks. To resolve the system virtual energy updating problem brought by the warping mechanism, we can either modify the way system virtual energy is computed, or modify the way the actual starting tag of newly joined tasks is updated.

4 Test-bench for Simulation

This chapter introduces the SystemC test-bench utilized to simulate and assess our proposed algorithms. The SystemC development environment is set up on Eclipse Galileo over Ubuntu 10.04, and the codes are compiled using the gcc 4.4.3 compiler with the support of the SystemC 2.2.0 library.

To begin with, the general structure of the test-bench is given. The function of each module, the way how different types of task are modeled and how different modules interact with each other are introduced. Then, the SystemC design of our algorithms is described in detail based on the design flow chart of BSEFQ. Finally, we introduce how extra SystemC codes are executed to obtain scheduling results regarding proportional power sharing, deadline missed by the real-time task, and response time of interactive task.

4.1. The Structure

Figure 4. 1 shows the structure diagram of the test-bench. The design is based on the producer-consumer model. Generally there are three modules: the producer, the fifo channel and the consumer, which are implemented in C++ header files producer.h, rq_fifo.h and consumer.h, respectively. A task is an instantiation of the producer module. It generates the energy requests in the form of service quanta[9] and writes them into its request queue, which is an instantiation of the fifo channel with sufficient length. The scheduler is an instantiation of the consumer module. Based on the scheduling policy, it selects the next service quantum to serve from the candidate request queues and updates the task virtual energy and system virtual energy according to the size of the selected energy packet. The main.cpp is in charge of assigning the initial shares and weights, instantiating the producer, fifo channel and consumer, and connecting the different instantiations. For simplicity, the duration of each service quantum is set to its maximum length Q and normalized to 1 CPU time unit (Tu); and the length of the period of any periodical task is set to be an integral multiple of the length of service quanta. For example, if a task has a period of 10 Tus and in each period requests 3 service quanta that lasts 3 Tus, its workload is 30%. Since each service quantum lasts one CPU time unit, only the value of the energy packet size is written into the request queue through a write( ) function. The write( ) function is declared in an interface class named rq_fifo_out_if, defined in rq_fifo.h, and implemented in the output port of an task. Similarly, the scheduler only reads the size of a selected energy packet through a read( ) function, which is declared in an interface class named rq_fifo_in_if, defined in rq_fifo.h, and implemented in the input port of the scheduler.

For periodical time-sensitive tasks that generate energy requests periodically, the producer together with the main.cpp control the length of the period, the number of service quanta per period, and the size of energy packet. Real-time task has fixed-length periods, while interactive task has unfixed-random-length periods. In both periodical real-time and interactive tasks, the number of service quanta per period as well as the energy packet size are variable, and the set of service quanta in each period are simultaneously requested in the beginning time of the period. If the service quanta of a periodical task are consumed faster than they are generated, that is to say

[9] Time quantum and energy packet are collectively called as service quantum.

the work of one period is finished before the end of the current period, then the corresponding request queue will become empty before the beginning of the next period, during which the task is regarded as an idle task. For batch tasks that continuously generate energy requests, the producer generates an energy request and writes it into the request queue every CPU time unit, which turns out to the scheduler that a new service quanta is requested once its previous service quantum is finished. Therefore the request queue will never become empty until all the work is finished. Like in the real-time tasks and interactive tasks, the energy packet size of batch tasks is also variable. All variable numbers are generated from the C++ random function with a discrete uniform distribution. Thus, their average values can be easily obtained for computing the maximum long-term power share of periodical time-sensitive tasks.

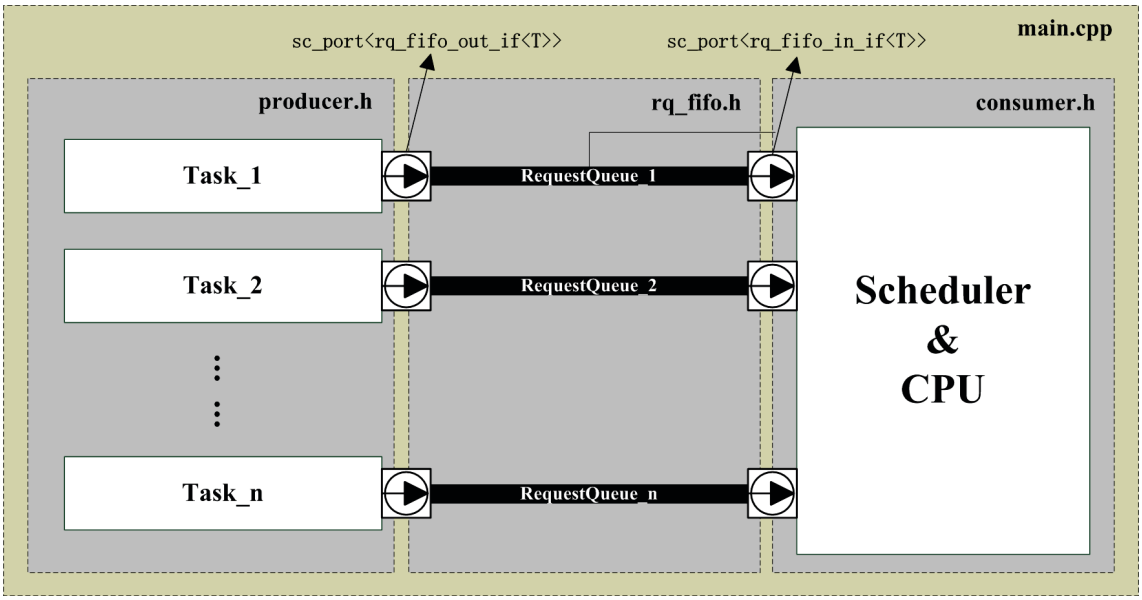


Figure 4. 1 Structure diagram of the test-bench

4.2. SystemC design of the scheduling algorithms

This section describes the SystemC design of the scheduling algorithms in detail; a flow chart for the BSEFQ is given in Figure 4. 2. Since SEFQ can be seen as a special case of BSEFQ with all the tasks un-warped, the flow chart for SEFQ is not specifically given. In the current test-bench design for BSEFQ, we use two functions of system virtual energy $V_{RT}(t)$ and $V_{Reg}(t)$ to respectively track the virtual energy advancing of time-sensitive tasks and regular tasks.

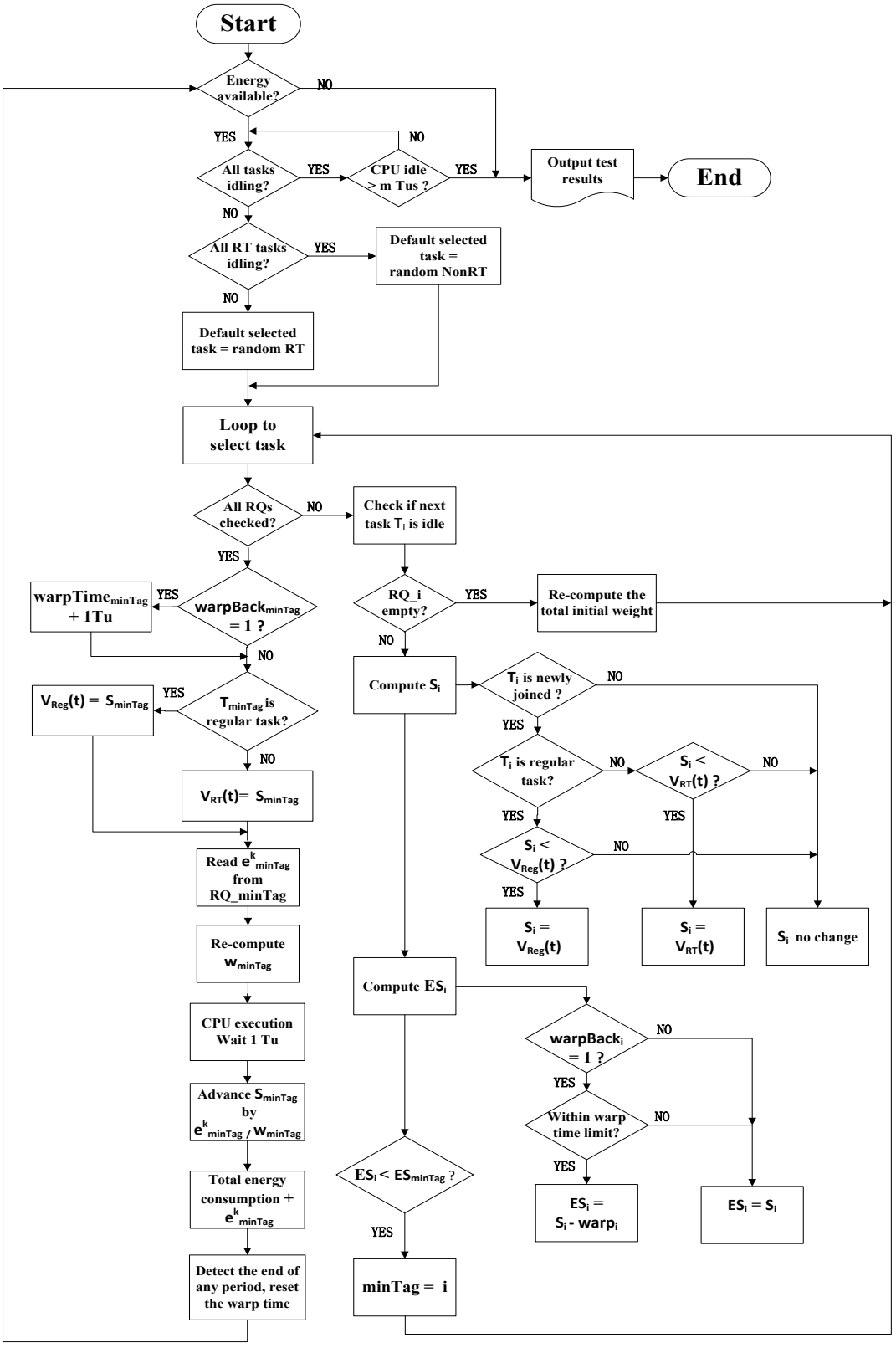


Figure 4. 2 Flow Chart of BSEFQ

To begin with, the scheduler checks if there is energy available for consuming. This is achieved by incrementing the total energy consumption at the end of each

scheduling loop and comparing that with the total energy available in one epoch E^{i}_{epoch}. If all the energy in one epoch is exhausted, the program outputs the test results in txt files and quits the execution. If energy is still available, we further check if all the tasks are idling by checking if all the request queues (RQ) are empty. To check if a request queue is empty, we use a function named get_num_request_left() to return the number of values available in the fifo channel. If it returns zero, the request queue is empty and the task is regarded as idle. If all the tasks are continuously idle (correspondingly the CPU is idle) more than a certain number of CPU time units, the program outputs the test results and stops.

The next step is to select a default dispatching task for tie-breaking when two tasks has the lowest effective starting tag at the same time. We first select the default dispatching task randomly among the real-time tasks. A random non-real-time task is selected only if all the real-time tasks are idle. A tie-breaking that favors real-time tasks can better support real-time performance without affecting the fairness. Each task has an ID, and if one task is selected as the default dispatching task, its ID value is assigned to a variable named smallestVSE_tag (appears as minTag in the flow chart)that is created to store the ID of the task with the lowest virtual starting tag.

After the default dispatching task is randomly determined, a loop is executed to compare the effective starting tag of all active tasks to select the one with the lowest effective starting tag to be dispatched next. The loop is executed a number of times that equals to the number of all instantiated tasks (active and non-active), and in each loop one task is involved. In the loop, we first check if one task is idle by checking its request queue. Once an idle task is detected, the total initial weight is recomputed by subtracting the initial weight of the idle task. For an active task, we further check if it is a newly joined task that has just made its transition from idle state to active state. If a newly joined time-sensitive task is detected and its actual starting tag S_i is lower than $V_{RT}(t)$, the current system virtual energy for time-sensitive tasks, the actual starting tag S_i is updated to the value of $V_{RT}(t)$. If the newly joined task is a regular task with its actual starting tag S_i lower than $V_{Reg}(t)$, the current system virtual energy for time-sensitive tasks, the actual starting tag S_i is updated to the value of $V_{Reg}(t)$. Next step computes the effective starting tag ES_i. For a warp-enabled time-sensitive task ($warpBack_i = 1$) that is within its warp time limit, its effective starting tag ES_i is the actual starting tag S_i subtracts its warp value $warp_i$; otherwise, the effective starting tag ES_i equals to the actual starting tag S_i. Based on the effective starting tag, the current task is compared with the task whose ID is stored in the variable smallestVSE_tag. If it

has a lower effective starting tag ($ES_i < ES_{minTag}$), the ID value of the current task T_i is assigned to the variable smallestVSE_tag. When the loop finishes with all the request queues checked, we have selected the task T_{minTag} with the lowest effective starting tag and updated the total initial weight.

Once the next dispatching task T_{minTag} is selected, the following steps are executed regardless of the order before the CPU execution. First, if the selected task T_{minTag} is a warped task ($warpBack_{minTag} = 1$), its warp time is incremented by one CPU time unit. Second, if T_{minTag} is a regular task, $V_{Reg}(t)$ is updated to the value of the actual starting tag S_{minTag} of the selected task; otherwise, T_{minTag} is a time-sensitive task, $V_{RT}(t)$ is updated to the value of the actual starting tag S_{minTag}. Third, the size of the selected energy packet e^k_{minTag} is read from the selected request queue. Forth, since complete idle of time-sensitive tasks is not considered within one epoch, the effective weight w_{minTag} of the selected task are computed according to equation (3. 9). Then, the CPU execution of the selected service quantum is simulated by waiting one CPU time unit. After that, the actual starting tag S_{minTag} of the selected task is advanced by adding to itself the normalized energy $\frac{e^k_{minTag}}{w_{minTag}}$, which is the size of the selected energy packet e^k_{minTag} divided by its recalculated effective weight w_{minTag}. And the total energy consumption is incremented by adding the size of the selected energy packet e^k_{minTag}, it will be used to check if the total energy consumption in the system has reached E^i_{epoch} in the beginning of the next scheduling loop. At the end of the loop, based on the current time in the CPU, if the end of the period of any periodical task is detected, the warp time of the task is reset to be zero and the time-constraint meeting is checked. Also, the amount of energy received by each task is sampled into a text file to check if tasks consume energy proportionally to their allocated power shares.

4.3. Scheduling results testing

In this section, we introduce how extra SystemC codes are inserted into the consumer.h to test the scheduling results. This section is divided into three sub-sections according to different concerned issues: proportional share of energy among different tasks, number of deadlines missed by the real-time task, and response time of interactive task. The performance of batch task is measured implicitly by its proportional share of energy.

4.3.1 Proportional power sharing

Figure 4. 3 shows the SystemC code employed to check the proportional power sharing among different tasks. This code is added at the end of the scheduling loop to check if tasks can consume energy proportionally to their allocated power shares. Every 100 Tus, the current CPU time, the cumulative energy consumed by each task, and the power during the previous 100 Tus are sampled into a txt file through an ofstream function named out_statistics. We obtain the current CPU time in the simulation by using the sc_time_stamp() function, and convert the basic CPU time unit into 1ms. The cumulative_served_energy array tracks the cumulative energy consumed by each task from the beginning of simulation. The power_Array stores the total energy consumed by each task in the previous 100 Tus, so it is reset to zero for the next 100 Tus.

```
/* output the statistical data every 100 Tus for check proportional share of energy*/
if((int(sc_time_stamp().to_seconds()*1000))% (100)==0){
    out_statistics << sc_time_stamp().to_seconds()*1000 << "\t";
    for(int k=0;k<num_application;k++){
        out_statistics << cumulative_served_energy[k] << "\t";
    }

    for(int k=0;k<num_application;k++){
        out_statistics << power_Array[k]/100 << "\t";
        power_Array[k] =0;
    }
    out_statistics << endl;
}
```

Figure 4. 3 SystemC code for checking proportional power shares

4.3.2 Deadline meeting

Figure 4. 4shows the SystemC code for checking the deadline missed by real-time task. The first step is to detect the deadline or the end of a period. The array join_time stores the time when a periodical real-time task joins the system, and the array period stores the period length of the real-time task. The cumulative available time[10] of a real-time task is obtained by using the current CPU time subtracts its join time, a deadline is detected if the cumulative available time is an integer multiple of the

[10] A real-time task is available from the time it joins the system to the time it completely leaves the system, a periodical task leaves the system temporarily before the beginning of the next period is still considered as available.

period. Since any unfinished service quantum in one period is postponed to the latter periods, we use the expected cumulative service time instead of the expected service time in each period to detect the missed deadlines. Once a deadline is detected, we compare the actual cumulative service time and the expected cumulative service time at the deadline, if the former is lower than the latter, a deadline is missed and the number of missed deadline is incremented by one. The expected cumulative service time is stored in the array cumulative_served_TQ_expected, and it is incremented by adding the number of service quanta in each period. The number of service quanta in each period is stored in an array pointed by pointer addressDC[i], which is passed from the main.cpp when the consumer is instantiated. At the end of this code, the pointer is incremented by one to point to the number of service quanta in the next period. Also, the cumulative warp time is reset to be zero so that the real-time task can run warped again in the next period.

```
// check the deadlines missed by real-time tasks
if(int(sc_time_stamp().to_seconds()*1000 - join_time[i]) % period[i]==0){
    cumulative_served_TQ_expected[i]+= (*addressDC[i]);
    if ((process_state[i]==1)&(cumulative_served_TQ[i] < cumulative_served_TQ_expected[i])){
        deadline_missed_Array[i]++;
    }
    addressDC[i]++;
    cumulative_warp_time[i]=0;
}
```

Figure 4. 4 SystemC code for checking deadline missed by real-time task

4.3.3 Response time

Figure 4. 5 shows the SystemC code for checking the response time of interactive tasks. The first piece of code records the response time of energy serving in each period, while the second piece of code computes the average and maximum response time for interactive task at the end of the simulation.

The first step is to detect the time when the set of service quanta in the former period are finished serving. The variable interactionForPC[i] stores the number of times one interactive task has been scheduled, that is, the number of service quanta already been served to the interactive task. The pointer addressIntEnergy points to the number of service quanta requested by the interactive task in each period. Each time the interactive task receives exactly the same number of service quanta as requested in the current period, we keep the current CPU time as the service finishing time of the current period, store it in an variable named serviceFinishingTime, and measure the response time for receiving all the service quanta from the time they are actually

requested, that is, the beginning time of the period stored in the variable requestTimeInteractive_actual. The response times of different periods are stored in the array responseInteractive. We use another variable requestTimeInteractive_ideal to store the ideal time that the set of service quanta in each period are issued, and it is incremented by adding the length of each period (or interval) of the interactive task. The length of each period is stored in an array pointed by addressInterval, which is passed from the main.cpp when the consumer is instantiated. Since the interactive task only starts a new period on condition that the service quanta from the previous period have been finished execution, the actual request time requestTimeInteractive_actual is computed in two cases. If the service finishing time of the former period is larger than ideal request time of the current period, it means the actual request time of the current period should be delayed to the moment when the former set of service quanta are finished serving. Otherwise, the actual request time of the current period is the same as the ideal one. After recoding the response time, both the pointers addressIntEnergy and addressInterval are incremented by one to point to the next period; the variable interactionForPC[i] is reset to zero to recount the number of service quanta served to the interactive task; the cumulative warp time of interactive task is reset to zero; and also the number of periods that the interactive task has been executed is recorded in the variable num_period_Array[i] for computing the average response time at the end of the program.

```
// counting the periods Interactive task has been executed,record the response time in each period, and reset the warp time
if(interactionForPC[i]==*addressIntEnergy){
    requestTimeInteractive_ideal += *addressInterval;
    if(seriveFinishTime > requestTimeInteractive_ideal){
        requestTimeInteractive_actual = seriveFinishTime;
    }
    else{
        requestTimeInteractive_actual = requestTimeInteractive_ideal;
    }
    responseInteractive[num_period_Array[i]] = int(sc_time_stamp().to_seconds()*1000) - requestTimeInteractive_actual;
    seriveFinishTime = int(sc_time_stamp().to_seconds()*1000);
    addressIntEnergy++;
    addressInterval++;
    interactionForPC[i]=0;
    cumulative_warp_time[i]=0;
    num_period_Array[i] ++;
}

// when the program quits, calculate the average maximum response time of interactive task
maximunResponseInteractive = responseInteractive[0];
for (int k=0;k<num_period_Array[i];k++){
    totalResponseInteractive += responseInteractive[k];
    if(maximunResponseInteractive < responseInteractive[k])
        maximunResponseInteractive = responseInteractive[k];
    }
averageResponseInteractive = totalResponseInteractive/num_period_Array[i];
```

Figure 4. 5 SystemC code for checking the response time of interactive task

The second piece of code is executed to compute the maximum and average response time at the end of the simulation. Through a loop comparison, the maximum response time is found among the response time of all periods stored in the array

response Interactive. The average response time is obtained by dividing the sum of all response times with the number of periods executed.

4.4. Summary and limitations

In this chapter, we have designed a SystemC test-bench based on the producer-consumer model. In the test-bench, tasks work as producers to generate service quanta requests and store the value of energy packet size in their corresponding fifo channels known as request queues; while the scheduler works as a consumer to select from the candidate request queues the next service quantum to serve based on the scheduling policy. Based on the type of tasks, different models are employed to generate the service quanta. Also, we add extra codes to obtain the scheduling results regarding proportional power sharing, deadline missed by the real-time task, and response time of interactive task.

This test-bench can be employed to test and verify the logic correctness of our scheduling scheme design and provide feedback for improving our design. However, it is far from perfect and there are several shortcomings to be conquered. First, for simplicity and without affecting the functionality, the test-bench assumes that the length of all service quanta is the same and equals to one CPU time unit. With this assumption, it is impossible to further simulate more complex algorithms that require the length of service quanta to be variable. For example, if we combine the DVFS into the SEFQ, the length of service quanta has to change based on the frequency chosen in different moments. Second, our simulation is based within one epoch and does not consider dynamic task participations over different epochs. It is necessary to extend the simulation to multiple epochs for better assessing the scheduling algorithms, and also supporting the simulation of more complex schemes like the one combined with DVFS. Third, the values of energy packet size used in our simulation are not really meaningful energy or power values. They may fail to reflect the energy consumption of different real applications due to their great differences and variations. Once our instrumented platform is ready for use, energy consumption values based on real measurement can be utilized in the simulation to have meaningful energy scheduling results.

5 Experiments and Results

In this chapter, simulations are implemented to verify the properties of our algorithms; simulation results with different settings under SEFQ and BSEFQ are analyzed and compared.

To begin with, the parameters of tasks under simulation is introduced, based on that, the maximum long-term power share as well as the worst-case power share of time-sensitive tasks are computed. Then, the scheduling results on proportional energy use under SEFQ and BSEFQ are verified and analyzed. After that, we demonstrate how to extend the CPU active time to the whole epoch by restricting the energy allocation. Later, we analyze and compare the real-time performance under SEFQ and BSEFQ with different settings. At last, further simulations are implemented to verify how the power control and time-constraint meeting can be traded-off under BSEFQ.

5.1. Simulation parameters setting

In the simulation, a system with 630,000 energy units (Eus) in total and a target lifetime of 63,000 CPU time units (Tus) is considered; the total energy is evenly allocated to 10 epochs with the same length. Then, each epoch lasts 6,300 Tus, with an energy per epoch, E^{i}_{epoch}, of 63,000 Eus and an average power, P^{i}_{epoch}, of 10 power units (Pus). Since a simulation of the whole operational time is a repeating simulation of each epoch, all the analysis given in latter sections are based on the simulation results within one epoch.

Table 5. 1 lists the features of the set of tasks considered in the simulation. Real-time and interactive tasks are time-sensitive tasks that have a non-zero initial power share; Batch 1 and Batch 2 are regular non-real-time tasks with a non-zero initial weight. Real-time task has fixed-length periods of 7 Tus, while interactive task has random-length periods that range from 10 to 15 Tus. Both Real-time and interactive tasks have variable number of service quanta per period; and each energy packet has variable size. All variable numbers are generated from the C++ random function with a discrete uniform distribution. Thus, their average values can be easily obtained. Real-time task averagely has 3 service quanta per period and the average size of energy packets is 15 Eus. Interactive task has an average period of 12.5 Tus, every period averagely has 2.5 service quanta and the average size of energy packets is 5 Eus. The energy packet size of two batch tasks is also variable. However, for simplicity, the average size of energy packet in both batch tasks is set to be 10 Eus to ensure a constant maximum long-term power share and worst-case power share within the epoch.

	Real-time	**Interactive**	**Batch 1**	**Batch 2**
Period (Tus)	7	10-15	N/A	N/A
Average Period (Tus)	7	12.5	N/A	N/A
Number of service quanta / period	2-4	1-4	N/A	N/A
Average service quanta / period	3	2.5	N/A	N/A
Size of energy packet (Eus)	10-20	3-7	8-12	6-14
Average size of energy packet (Eus)	15	5	10	10
Maximum long-term power share	0.58	0.09	$x \to 1$	$x \to 1$
Practical worst-case power share	0.777	0.173	N/A	N/A

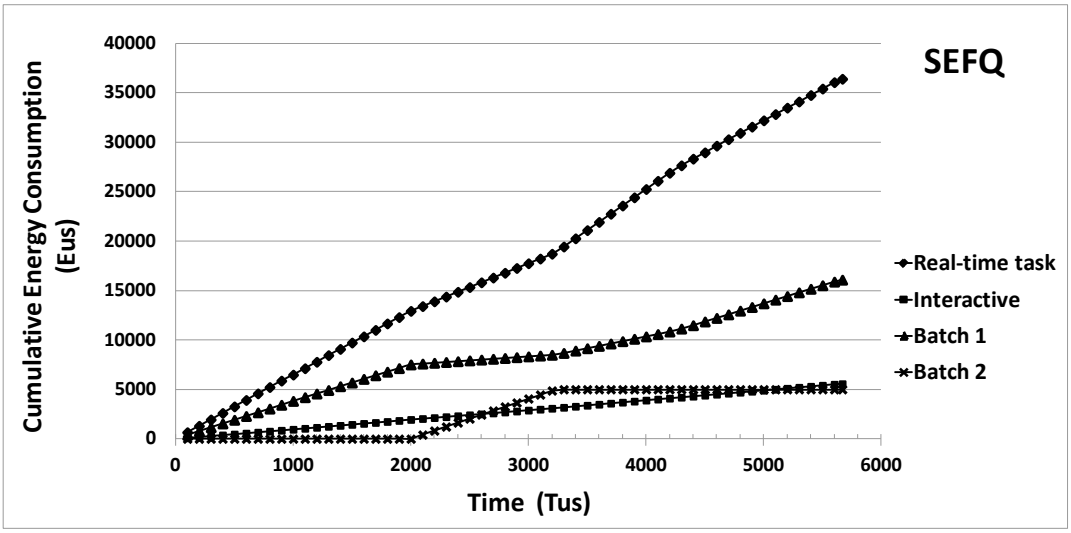
Table 5. 1 List of tasks in the simulation

Based on the parameters of different tasks, we compute the maximum long-term power share and practical worst-case power share. In the long-term, the maximum average power of real-time task is $\frac{3\times15}{7} = 6.43\ \mathrm{Pus}$, the maximum average power of interactive task is $\frac{2.5\times5}{12.5} = 1\ \mathrm{Pus}$, and two batch tasks together have an minimum average power of $\left(1-\frac{3}{7}-\frac{2.5}{12.5}\right)\times 10 = 3.71\ \mathrm{Pus}$. Therefore, the maximum long-term power share of real-time task and interactive task is 0.58 and 0.09, respectively. When real-time task has the worst-case power of $\frac{4\times20}{7} = 11.43\ \mathrm{Pus}$, the average power of interactive task is $\frac{2.5\times5}{12.5} = 1\ \mathrm{Pus}$, and the average power of two batch tasks is $\left(1-\frac{4}{7}-\frac{2.5}{12.5}\right)\times 10 = 2.29\ \mathrm{Pus}$, thus, the practical worst-case power share of real-time task is 0.777. Similarly, the practical worst-case power share of interactive task is 0.173.

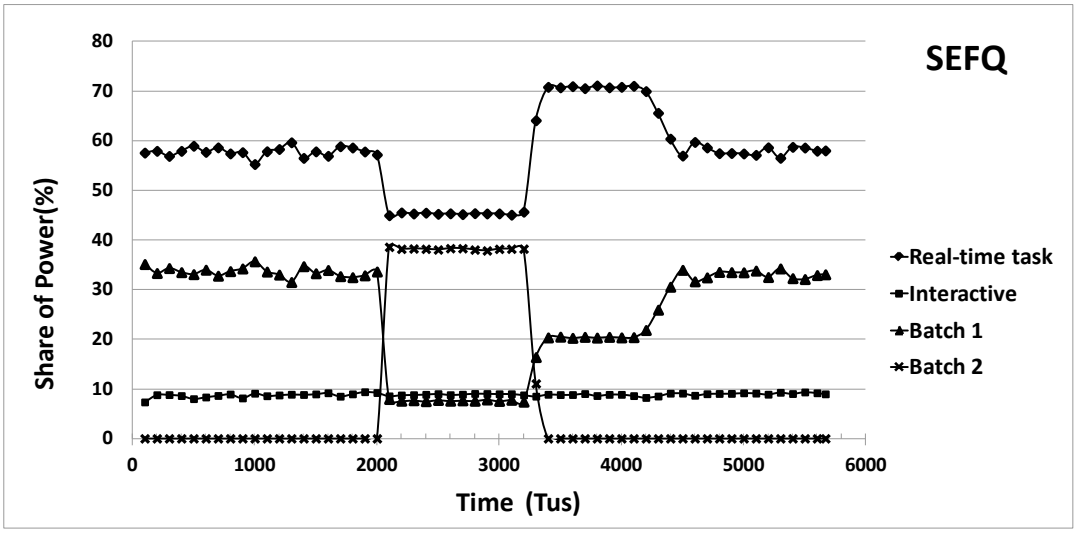
5.2. Proportional power sharing

Figure 5.1 shows the proportional share of energy among the four tasks under SEFQ. Figure 5.1(a) shows the cumulative energy consumption along the time while Figure 5. 1(b) shows the power shares. To reduce the impact of random numbers used in Table 5.1, all the graphs in this section are based on the averaging of 10 simulations with different random number seeds. The initial power shares of real-time, interactive, batch 1 and batch 2 are 0.3, 0.09, 0, 0, respectively, and their initial weights are 2, 0, 1, 5. Batch 2 is delayed 2,000 Tus, allowed to run until 5,000 Eus are received; all other tasks start at time 0 and run until exhausting $\mathrm{E}^{\mathrm{i}}_{\mathrm{epoch}}$.

In Figure 5.1(b), fluctuations are constrained to an average value, demonstrating that each task consumes energy with a power share that is long-term guaranteed. The sharp slopes in real-time, batch 1 and batch 2 are due to the join and leave of batch 2 task, however, the power share of real-time task is guaranteed no lower than 0.3 and the one of interactive task is guaranteed no lower than 0.09. Since 0.09 is also the maximum long-term power share of interactive task, during the whole simulation interactive task receives a constant long-term power share that is not affected by dynamic task participation. Initially batch 2 is idling, the effective power shares and effective weights of real-time, interactive and batch 1 are 0.707, 0.09, 0.203 respectively. Real-time task receives an extra power share of 0.307 with its initial weight. However, since the maximum long-term power share of real-time task is 0.58, real-time task will release its excess long-term power share of 0.127 and reallocate it to



(a)



(b)

Figure 5. 1 Proportional energy use under SEFQ with maximum long-term power share not guaranteed

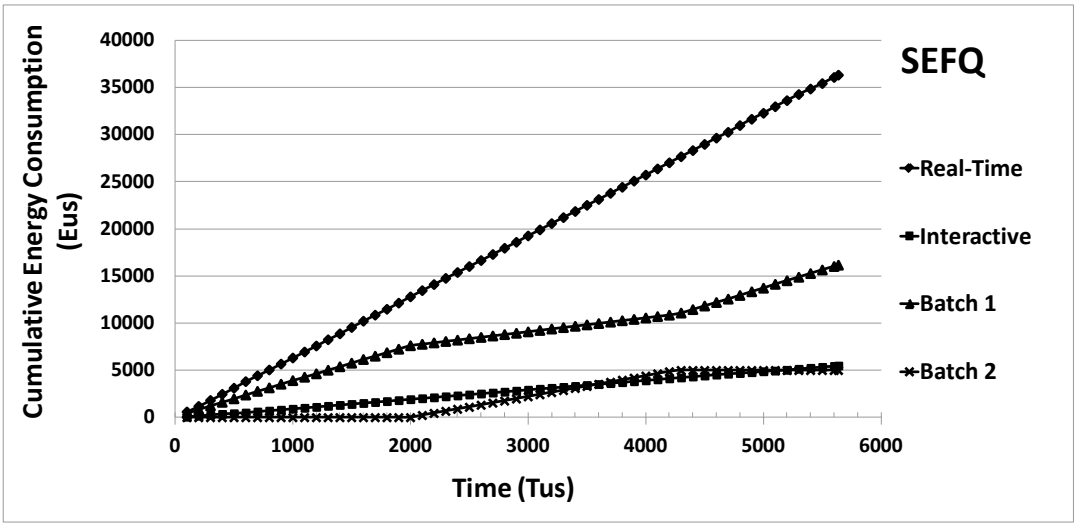
batch 1 when finishes its work and goes to idle temporarily. Therefore, the actual long-term power shares are 0.58, 0.09 and 0.33. At time 2000, batch 2 joins the competition with initial weight being 5, the effective power shares and weights are recomputed as 0.45, 0.09, 0.08, and 0.38. In this case, real-time task receives an extra power share of 0.15, the actual power share is also 0.45 because it is lower than the maximum long-term power share. At time around 3200, batch 2 becomes completely passive and leaves the system after receiving 5,000 energy units, thus, the effective power shares and effective weights of real-time, interactive and batch 1 are again recomputed as 0.707, 0.09, 0.203. However, in this case the actual power share of real-time task

keeps as 0.707 for a period of time before it falls down to the maximum long-term power share 0.58. This is caused by the delayed service quanta that are not finished in the previous interval, during which the power share 0.45 is less than the maximum long-term power share. At time around 4400, the actual long-term power share of real-time task falls back to the maximum long-term power share 0.58, since interactive task cannot take more power share allocation, the released long-term power share of 0.127 is reallocated to batch 1. At time around 5653, the CPU is forced to idle before the end of the epoch due to the exhausting of E_{epoch}^{i}.

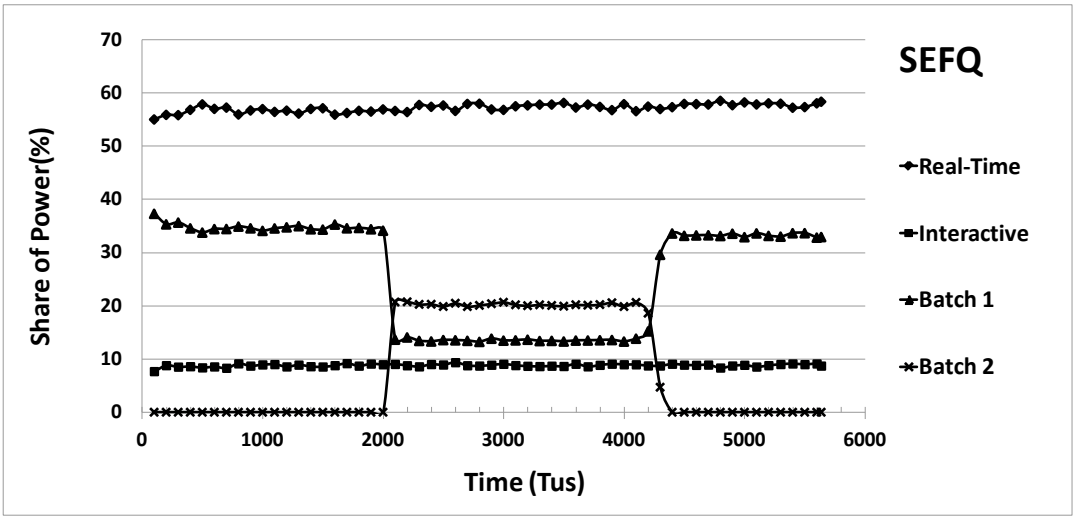
During the whole simulation in Figure 5. 1, interactive task has a constant power share of 0.09 $P(t)$, thus, consumes 0.09 share of the E_{epoch}^{i}, that is 5,670 Eus in total, then the corresponding average power over one epoch T_{epoch}^{i} is 0.9 Pus, which means a 0.09 share of P_{epoch}^{i} is guaranteed to interactive task. The power share of real-time task is protected to be no lower than $0.3P(t)$, so real-time task is guaranteed a 0.3 share of P_{epoch}^{i}. In this simulation, since all delayed service quanta of real-time task are served before the CPU goes to idle, its average power share of $P(t)$ during the whole CPU active period equals to the maximum long-term power share 0.58, which means averagely a 0.58 share of P_{epoch}^{i} is allocated to the real-time task. The shares of P_{epoch}^{i} allocated to batch 1 and batch 2 tasks are computed in a slightly different way due to the dynamic task participation during the execution. For batch 2, since it receives 5,000 energy units in total, the average power over one epoch T_{epoch}^{i} is 0.8 Pus, thus 0.08 share of P_{epoch}^{i} is allocated to batch 2. The remaining 0.25 share of P_{epoch}^{i} is allocated to batch 1.

The scheduling results of Figure 5. 1 illustrate that SEFQ can proportionally allocate energy among tasks. However, the power distribution along the time is poor since real-time task is not always guaranteed at least the maximum long-term power share 0.58. Between time 2000 and 3200, the service quanta of real-time task are severely delayed and miss their deadlines due to the power restriction, while after time 3200, the delayed service quanta have to be executed with a higher power. The power share variation of real-time task increases the difficulty in power control and brings challenge in time-constraint meeting. To solve the problem, we can either increase the allocated power share or degrade the quality of real-time application.

Figure 5. 2 shows the proportional power sharing under SEFQ when both real-time task and interactive task are guaranteed their maximum long-term power shares. In this simulation, the initial power shares assigned to real-time task and interactive task are 0.58 and 0.09, respectively; the initial weights assigned to batch 1 and batch 2 are 2 and 3, respectively. We do not assign a higher initial share or a non-zero initial weight to time-sensitive tasks because in the long-term the average power share is no greater than the maximum long-term power share, any over-allocated power share will be released and reallocated to the two batch tasks. Therefore, when real-time task and interactive task are assigned higher initial shares than their maximum long-term power



(a)



(b)

Figure 5. 2 Proportional energy use under SEFQ with maximum long-term power share guaranteed

shares, the actual long-term power shares among tasks are the same as shown in Figure 5. 2. Anyway, a higher effective share helps to meet more time constraints, which will be shown in the next section.

As shown in Figure 5. 2, both real-time task and interactive task receive constant long-term power shares of 0.58 and 0.09, respectively, as guaranteed by their initial shares. Batch 1 and batch 2 compete for the remaining 0.33 power share with their initial weights. Initially, batch 1 receives the whole 0.33 power share left by real-time class. At time 2,000, batch 2 joins the competition, the effective weights of batch 1 and batch 2 are recalculated to be 0.132 and 0.198, respectively, and thus, batch 2 is allocated a power share of 0.198 while the power share of Batch 1 drops to 0.132. At time quantum around 4,300, batch 2 becomes completely passive and leaves the system after receiving 5,000 energy units, thus, batch 1 regains the whole 0.33 power share. At time 5,637, CPU goes to idle before the end of the epoch due to the exhausting of $\mathrm{E}_{\mathrm{epoch}}^{\mathrm{i}}$.

Figure 5. 3 shows the scheduling results under BSEFQ when both real-time task and interactive task are guaranteed the maximum long-term power share. The warp-time limit of each time-sensitive task is set to be larger than the average number of service quanta per period. For example, the warp time limit of real-time task can be 3 or 4 Tus. As can be seen, the power sharing among different tasks is similar to Figure 5. 2; however, the slightly larger fluctuation under BSEFQ indicates a worse fairness bound.

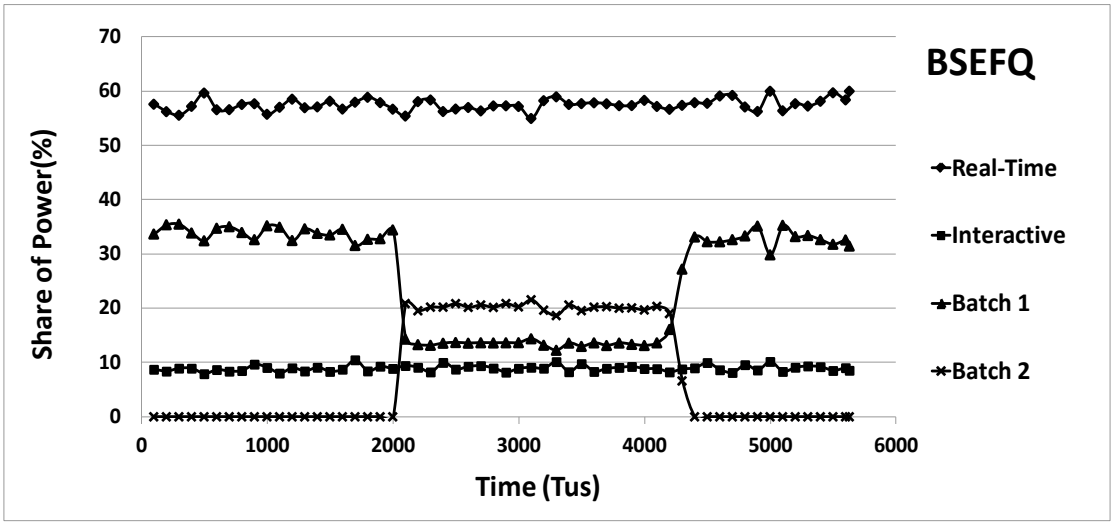
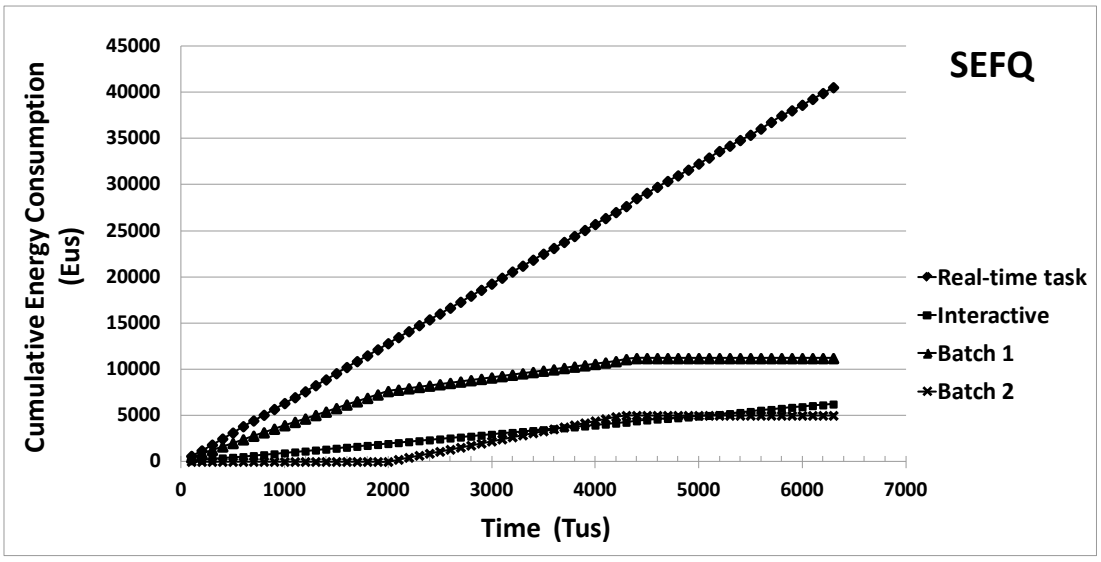


Figure 5. 3 Proportional energy use under BSEFQ with maximum long-term power shares guaranteed

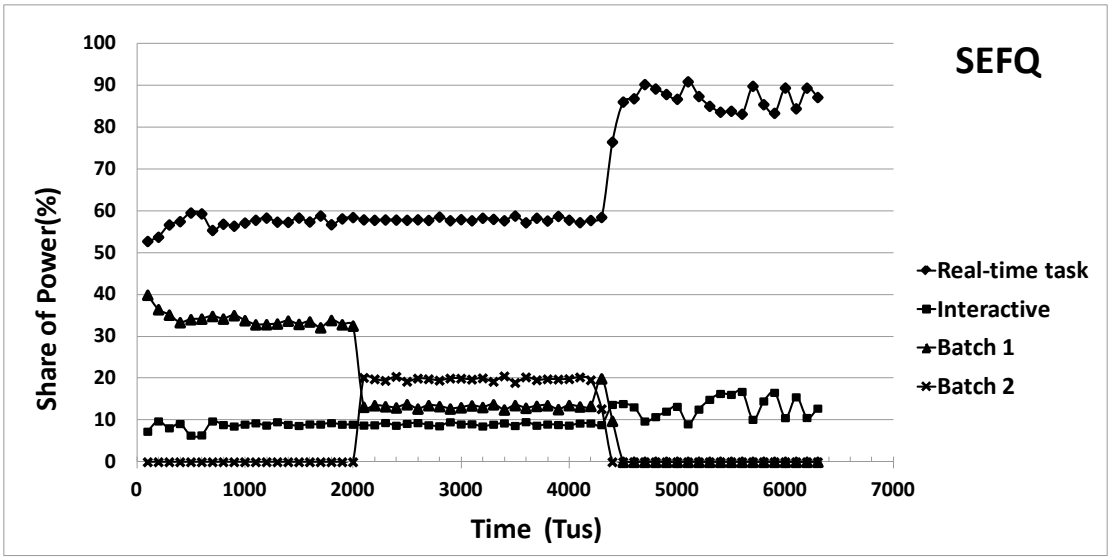
5.3. Extend task execution to the whole epoch

As shown by all the three figures in section 5.2, the CPU is forced to idle before the end of each epoch. Note that the idle interval does not always appear, depending on the average power of active tasks and their effective power shares, the CPU may be active over the whole epoch or has a variable-length idle interval. It can be seen as a result of the mismatch between an ideal system assuming constant CPU power and a real system with variable CPU power. A variable-length CPU idle interval produces bursty behavior between epochs, which is not tolerated by periodical hard real-time tasks, and also not appreciated by those applications that need to provide a smooth user experience, such as multimedia applications and interactive applications.

To extend the CPU active time to the whole epoch, we can restrict the energy allocation of those batch tasks to guarantee adequate energy for real-time and interactive tasks. Actually in the former simulations, we have already restricted the energy allocation of batch 2 to be no more than 5,000 Eus. In this simulation, we extend the CPU active time to the whole epoch by further restricting the energy allocation of batch 1 to be no more than 11,200 Eus. As shown in Figure 5. 4, batch 2 and batch 1 are forced to leave the energy competition at time around 4200 and 4300, respectively, which leaves just enough energy to support the execution of real-time and interactive task until the end of the epoch. When both batch 1 and batch 2 are idling, the power shares of real-time task and interactive task rise as shown by Figure 5. 4(b). However, their actual powers keep the same as shown by the slopes in Figure 5. 4(a), which indicates a reduced CPU power. This is because the CPU is free without executing any batch task when real-time and interactive tasks are temporarily idling. Figure 5. 4 is based on the simulation results of only one set of random numbers, which explains the more obvious fluctuation of power shares in Figure 5. 4(b).



(a)



(b)

Figure 5. 4 CPU active time extended to the whole epoch under SEFQ

Note that, the energy allocation of batch 1 has to be properly restricted to ensure the total energy E^{i}_{epoch} to be exhausted just at the end of one epoch. And, like the length of the CPU idle interval, the required energy restriction of batch 1 also varies depending on the average power of active tasks and their effective power shares. Figure 5. 5 shows the relationship between the energy allocation of batch 1 and the total CPU active time. As can be seen, if we overly restrict the energy allocation of batch 1 (implicitly lose the chance to improve the performance of batch 1), the total energy E^{i}_{epoch} cannot be exhausted at time 6300, indicated by a CPU total active time

longer than 6300 Tus; in contrast, if the energy restriction of batch 1 is too loose, the CPU still goes to idle before time 6300 due to the exhausting of the total energy E^{i}_{epoch}.

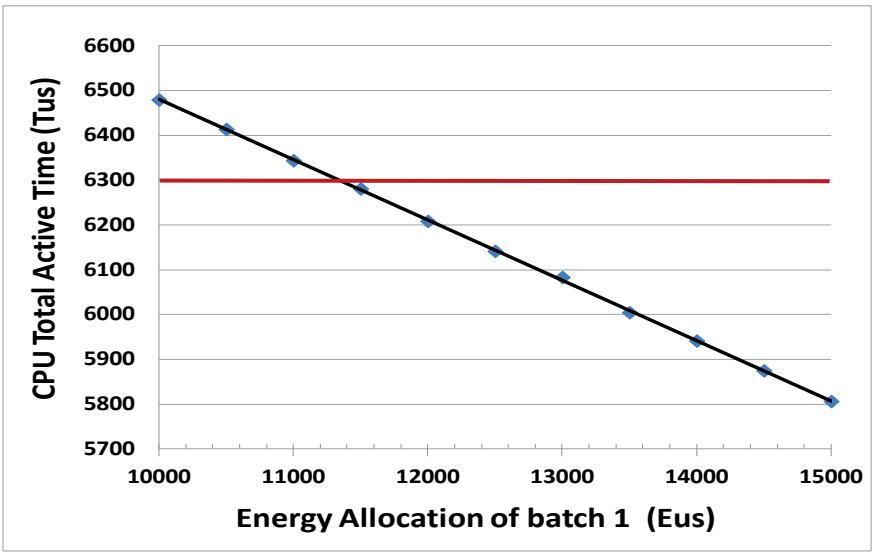


Figure 5. 5 Relationship between batch 1 energy allocation and total CPU active time

In this simulation, we extend the execution of real-time and interactive applications to the whole epoch by restricting the energy allocation of batch applications and thus, reduce their performance. The performance of the target applications are guaranteed at the cost of a reduced performance of those less important applications. However, in the real world, it is also possible that we restrict the energy allocation of several real-time or interactive applications to guarantee energy for the most important task. In this case, since a larger idle interval is introduced to those energy-restricted time-sensitive applications, their performance can be pretty poor, which affects the total applications performance. To maximize the user experience, the target applications can degrade their performance by self-reducing their energy demand and leave more energy for improving the performance of other applications. Moreover, those energy-restricted time-sensitive applications can also self-reduce their energy demand to extend their total execution time to approach the end of epoch.

The CPU idle interval in an epoch-combined SEFQ is similar to the one introduced by the DPM, in which the CPU finishes the work of all tasks in its maximum speed and goes to idle to save energy. The difference is that, in the epoch-combined SEFQ, the CPU is forced to idle due to the exhausting of limited energy; while in the DMP, the CPU goes to idle automatically due to the finishing of work. Correspondingly, one potential solution to the variable idle interval is to apply the DVFS to the epoch-combined SEFQ, but adjust the CPU voltage and speed based on the epoch.

5.4. Meeting time-constraint

Table 5. 2 compares the performance of time-sensitive tasks under SEFQ and BSEFQ with different number of batch tasks competing for the resource. In Table 5. 2(a), real-time and interactive task compete against 2 batch tasks, while in Table 5. 2(b) they compete against 8 batch tasks. $BSEFQ_1$ favors real-time task by setting the warp value of real-time task larger than the warp value of interactive task, while $BSEFQ_2$ favors interactive task by doing the opposite setting. In $BSEFQ_3$, real-time task and interactive task are given the same priority. In all BSEFQ simulations, the warp-time limit of both real-time task and interactive task is set to be 4 Tus, the maximum length of service time in both time-sensitive tasks, which means that the maximum power of a time-sensitive task is not restricted under BSEFQ. In section 5.5, we will show how the warp-time limit can be properly adapted to trade off the power control and time-constraint meeting. Real-time task has a total number of 803 deadlines on average. Any service quantum that misses its deadline is postponed to the latter periods. Thus, the deadline of one period may be missed even if the energy demands in that period are met. Interactive task starts a new period only on condition that service quanta from the previous period have been finished serving. To ensure a constant maximum long-term power share and worst-case power share for time-sensitive tasks, the average size of energy packet in all batch tasks is set to be 10, although the range and seed to generate random sizes are different. In this simulation, all batch tasks are assigned a initial share of 0 and a initial weight of 1, start at time zero and run until exhausting E^{i}_{epoch}. The initial shares for real-time task and interactive task are as listed in Table 5. 2, and their initial weights are all set to be zero.

The scheduling results listed in Table 5. 2 are based on the statistics of 40 sets of simulation data, generally there are four parts: the average value, the standard deviation, the confidence level (in the parentheses), and the multiplication coefficient that depends on the confidence level chosen. Those data without the confidence level specifically marked are holding a 68% confidence by default, and those data that only contains the average value are constant ones without any deviation.

In the first simulation under SEFQ ($SEFQ_1$), real-time task and interactive task are both assigned initial shares that equal to the value of their corresponding maximum long-term power shares 0.58 and 0.09, respectively. Although in the long-term all their energy demands are met, the real-time performance is very poor: real-time task misses majority of its deadlines and interactive task has long response time. For real-time task,

the number of periods in which the energy demands are not met is actually smaller. However, since unfinished service quanta are postponed to the latter periods, the number of missed deadlines turns out to be much larger. When there are more batch tasks competing energy in the system, the average number of deadline missed by real-time task and the maximum response time of interactive task are slightly larger, but it is hard to tell the difference in real-time performance due to the large deviations.

	Initial share real-time task	**Initial share interactive task**	**Warp value**	**Number of deadlines missed**	**Mean response time (Tus)**	**Max. response time (Tus)**
$SEFQ_1$	**0.58**	**0.09**	N/A	688.4±63.5	11.78±0.60	25.83±0.81
$SEFQ_2$	**0.777**	**0.09**	N/A	0.15±0.43*3.6 (99.9%)	5.57±0.21	16.15±1.14
$SEFQ_3$	**0.58**	**0.173**	N/A	233.6±31.1	4.80±0.18	12.25±0.71
$SEFQ_4$	**0.777**	**0.173**	N/A	2.45±1.78*1.7 (90%)	3.86±0.12	10.53±0.55
$BSEFQ_1$	**0.58**	**0.09**	real-time > interactive	0.00	4.21±0.12	11.93±0.27
$BSEFQ_2$	**0.58**	**0.09**	real-time < interactive	34.83±8.24	2.49±0.06	4.00
$BSEFQ_3$	**0.58**	**0.09**	real-time = interactive	0.03±0.16	3.80±0.12	11.00±0.51

(a) Competing with 2 batch tasks

	Initial share real-time task	**Initial share interactive task**	**Warp value**	**Number of deadlines missed**	**Mean response time (Tus)**	**Max. response time (Tus)**
$SEFQ_1$	**0.58**	**0.09**	N/A	694.4±62.9	11.75±0.67	26.43±1.26
$SEFQ_2$	**0.777**	**0.09**	N/A	13.63±2.98*3.6 (99.9%)	5.85±0.22	18.15±1.56
$SEFQ_3$	**0.58**	**0.173**	N/A	271.8±32.1	4.95±0.20	13.50±0.72
$SEFQ_4$	**0.777**	**0.173**	N/A	12.98±3.83*1.7 (90%)	4.04±0.14	11.30±0.72
$BSEFQ_1$	**0.58**	**0.09**	real-time > interactive	0.00	4.21±0.12	11.93±0.27
$BSEFQ_2$	**0.58**	**0.09**	real-time < interactive	34.75±8.28	2.49±0.06	4.00
$BSEFQ_3$	**0.58**	**0.09**	real-time = interactive	0.03±0.16	3.79±0.12	10.95±0.50

(b) Competing with 8 batch tasks

Table 5. 2 Comparison in performance of time-sensitive tasks

In the second SEFQ simulation ($SEFQ_2$), real-time task is assigned an initial share that equals to its practical worst-case power share 0.777, while interactive task keeps the initial share 0.9. In this case, real-time task nearly meets all the deadlines when competing with 2 batch tasks: there are only 1 or 2 deadlines missed in 5 out of 40 sets of simulation results, in the rest simulations all deadlines are met. Two reasons may lead to the few missed deadlines: first, the practical worst-case power share instead of the theoretical one is considered; second, the allocation error is not considered when computing the practical worst-case power share. Since the allocation

error of SEFQ increases linearly to the number of active tasks in the system, the performance of real-time task is obviously (99.9% confidence) much worse when competing with 8 batch tasks: the number of deadline missed ranges from 7 to 21 in our 40 sets of scheduling results, averagely 14 deadlines are missed. For interactive task, in both cases the mean response time and max response time is much shorter than those in $SEFQ_1$. This is because it benefits from the released power share when real-time task temporarily goes to idle due to the finishing of service before the end of each period. More specifically, when real-time task goes to idle, interactive task compete against all batch tasks with an effective weight of 0.09, while the sum of effective weights of all batch tasks is $1 - 0.09 - 0.777 = 0.133$. As a result, interactive task is assigned an effective share of 0.386 during the idle period of real-time task. Similar to $SEFQ_1$, the response time of interactive task is slightly larger when more active tasks are available in the system; however, the performance is not obviously worse due to the relatively large deviations.

In $SEFQ_3$, real-time task is assigned the initial share of 0.58 while interactive task is assigned an initial share that equals to its practical worst-case power share 0.173. With a great power share, the mean and maximum response time of interactive task is obviously shorter than those in $SEFQ_1$ and $SEFQ_2$. Since real-time task can benefit from the released power share of interactive task, it meets more deadlines than $SEFQ_1$. Again, averagely more deadlines are missed and the response time tends to be longer when there are more active tasks competing for the energy, but the difference in real-time performance is not obvious.

In the final simulation under SEFQ ($SEFQ_4$), both real-time task and interactive task are assigned initial shares that equal to their practical worst-case power shares. In comparison with $SEFQ_2$, when competing with 8 batch tasks, the performance of real-time task is close; however, when competing with 2 batch tasks, although real-time task can benefit from the released power share of interactive task, it tends to (68% confidence) miss slightly more deadlines due to the increased frequency that interactive task is scheduled. When both real-time and interactive task are competing for energy with high energy load, the more frequent interactive task is scheduled, the easier real-time task may miss its deadline during the competition with interactive task. Also, real-time task obviously (90% confidence) misses more deadlines when there are 8 batch tasks active in the competition. As far as the performance of interactive task is concerned, it has the shortest response time among all SEFQ simulations since interactive task can benefit from the released power share from real-time task.

In conclusion with the four SEFQ simulations, the allocation error should be taken into account when assigning real-time task a share to meet all its deadlines; the performance of interactive task, however, is not very sensitive to the allocation error. In SEFQ, since the allocation error of energy varies with the number of active tasks and is hard to be combined into the computation of worst-case power share, it is not sure how large a share should be reserved for one real time task to meet all its deadlines. For a real-time task requires all the deadlines to be strictly met, a conservative share that may be overly larger than the worst-case power share should be reserved to the task.

In comparison with SEFQ, the real-time performance under BSEFQ is totally insensitive to the allocation error. As shown in Table 5. 2(a) and Table 5. 2(b), the number of deadlines missed by real-time task and the response time of interactive task are almost the same when competing with different number of batch tasks. This is because by applying the warp mechanism, the time-sensitive tasks are completely isolated from the competition of batch tasks. Ideally, the scheduling results in two cases should be exactly the same; the slight difference is caused by the difference in the random values generated for energy packet size.

In $BSEFQ_1$, real-time task is assigned a higher warp value to allow all its service quanta to be scheduled immediately after a new period begins, thus, all deadlines are guaranteed to be met; the response time of interactive task is also short because its service quanta are continuously scheduled right after finishing the execution of real-time task, no service quanta of batch tasks is scheduled ahead of interactive task. In $BSEFQ_2$, since interactive task is scheduled with a higher priority than real-time task, interactive task achieves the optimal response time among all simulations while real-time task misses a small number of deadlines. In $BSEFQ_3$, the performance of real-time task and batch task is balanced by scheduling them with the same priority. Nearly all the deadlines of real-time task are met as in $SEFQ_2$, and the response time of interactive task is close to the optimal one under SEFQ ($SEFQ_4$). In all BSEFQ simulations, real-time task and interactive task are assigned the initial shares of 0.58 and 0.09 respectively, any short-term or small deviation on the value of initial shares does not seriously affect the real-time performance. In $BSEFQ_1$ and $BSEFQ_2$, since real-time task and interactive task are scheduled based on their priority, the initial shares are only used to keep track of the normalized energy received by time-sensitive tasks so that batch tasks can have the opportunity to get dispatched when all time-sensitive tasks are unwarped. In $BSEFQ_3$, we can trade off the performance of real-time and interactive task by changing their effective power shares, however, their

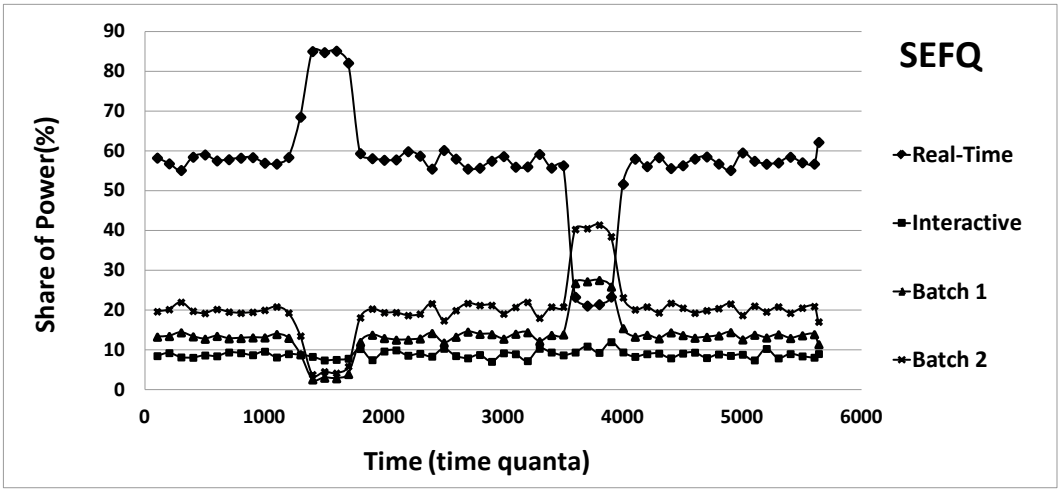
minimum real-time performance is guaranteed by the warp mechanism. The minimum performance of real-time task is guaranteed as in $BSEFQ_2$, and the minimum performance of interactive task is guaranteed as in $BSEFQ_1$. For any hard real-time task, a higher warp value can be assigned to strictly meet all its deadlines as in $BSEFQ_1$; for any soft real-time task, the warp value can be assigned based on its importance when compared with other time-sensitive tasks.

In conclusion, BSEFQ supports more stringent time-constraint meeting in a dynamic system while avoids the inconvenience to take the allocation error into account when computing the worst-case power share. Also, by combining priority-based scheduling, BSEFQ is more flexible and effective in supporting different types of time-sensitive tasks.

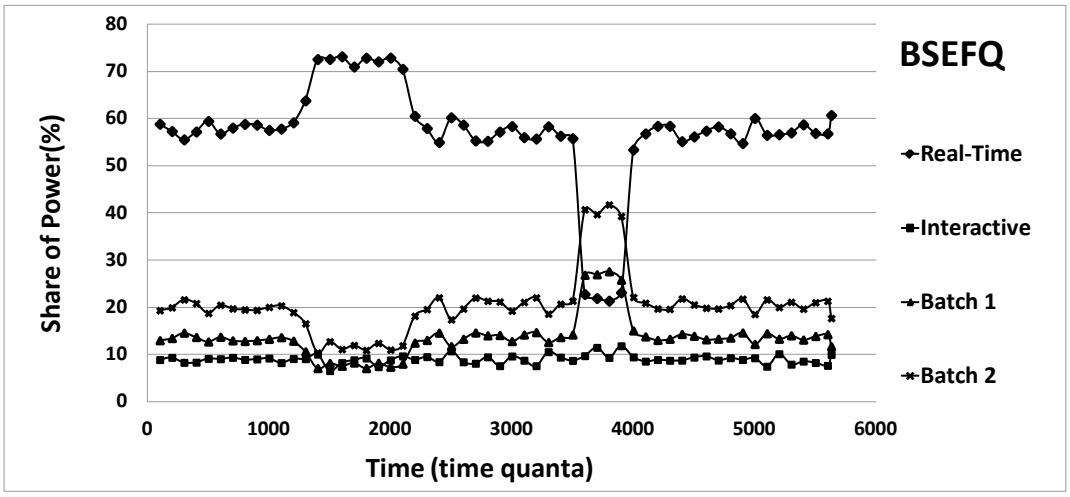
5.5. Trading off power control and time-constraint meeting

The simulation in this section shows how BSEFQ can flexibly trade off the power control and time-constraints meeting by properly adjusting the warp-time limit. The set of tasks considered for this simulation is the same as the one listed in Table 5. 1, except with some slight modifications for better comparing the results. In this simulation, the total energy allowed to be allocated to batch 2 task is not limited, all tasks start at time 0 and run until exhausting E^{i}_{epoch}. Between time units 1260 and 1680 (from its 180th period to 240th period), the workload of real-time task is incremented to 5 service quanta per period, leading to a practical worst-case power share of 0.852. To keep the average power of the real-time task the same as in Table 5. 1, between time units 3500 and 3920 (from 500th period to 560th period), the workload is reduced to be 1 service quantum per period. Therefore, the maximum long-term power share can be allocated to real-time task during this interval is 0.22, as can be seen in both Figure 5. 6(a) and Figure 5. 6(b).

Figure 5. 6(a) shows the proportional power sharing under SEFQ when real-time task is assigned an initial share that equals to its worst-case power share 0.85. Interactive task is assigned an initial share of 0.09, while batch 1 and batch 2 are assigned the initial weights of 2 and 3, respectively. The worst-case power share allows real-time task consuming as much energy as it demands, therefore, it takes a power share of 0.852 during interval 1260~1680 Tus. As a result, the power shares available to batch 1 and batch 2 are significantly reduced during that interval. Without



(a) SEFQ with worst-case power share guaranteed to real-time task



(b) BSEFQ with warp-time limit properly set

Figure 5. 6 Trading off power control and time-constraint meeting

setting warp-time limit, the power shares under BSEFQ are similar to those in Figure 5. 6(a). Figure 5. 6(b) shows the proportional power sharing under BSEFQ when the warp-time limit of real-time task is set to be 4 Tus. Since real-time task is allowed to run warped maximum 4 service quanta, the maximum power share of real-time task is restricted to be 0.72. As can be verified in Figure 5. 6(b), the power share allocated to real-time task is well controlled under 0.72 during interval 1260~1680 Tus, and unfinished service quanta of this interval is postponed to the latter periods, appears in Figure 5. 6(b) as a 0.72 power share that lasts longer than 420 time quanta. In both Figure 5. 6(a) and Figure 5. 6(b), the average power share of real-time task during the whole execution is 0.58. As far as the deadline meeting is concerned, real-time task misses 32 deadlines under SEFQ due to the energy allocation error, misses 129 deadlines under BSEFQ. Note that with a warp-time limit of 5 Tus for real-time task,

BSEFQ can meet all deadlines (regardless of the allocation error) at the cost of losing power control. In conclusion, by properly setting the warp-time limits under BSEFQ, time-sensitive tasks miss more deadlines when they are highly energy-loaded due to the power restriction, however, their energy consumption can be well-controlled to avoid bringing any energy starvation to the batch tasks. The warp time limit enables the trading off between power control and time-constraint meeting.

5.6. Summary

In the chapter, we have made several simulations to assess our algorithms. Simulation results show that SEFQ supports proportional power sharing among different tasks and the power share of any time-sensitive task can be protected to be no lower than its desired share in a dynamic system. In order to achieve stable and well-controlled power shares and better support periodical time-sensitive tasks, it is necessary to guarantee time-sensitive tasks their maximum long-term power shares. When multiple tasks consume energy in a high rate, the CPU may be forced to idle before the end of one epoch due to the exhausting of the total energy. By restricting the energy allocation of those less important applications, the CPU active time can be extended to the whole epoch for better supporting the target applications.

In SEFQ, the time-constraint meeting of real-time task is sensitive to the allocation error. However, the energy allocation error increases linearly to the number of active tasks, and it is difficult to be taken into account when computing the minimum power share required for time-constraint meeting. BSEFQ supports more stringent time-constraint meeting in a dynamic system while avoids incurring the inconvenient energy allocation error. The warp mechanism guarantees the real-time performance by giving scheduling priority to time-sensitive tasks, which breaks the fairness between time-sensitive tasks and regular tasks. However, in the long-term, the power shares of all tasks under BSEFQ are the same as those under SEFQ when the maximum long-term power shares of time-sensitive tasks are guaranteed. BSEFQ improves the performance of real-time task at the cost of fairness, which makes sense considering the conflict between maintaining a strict fairness and meeting time constraints. Furthermore, by setting different levels of warp values, BSEFQ can flexibly and effectively support various types of time-sensitive tasks. Finally, BSEFQ can trade off the power control and time-constraint meeting by properly setting the warp-time limits.

6 Conclusions and Future Research

6.1. Conclusions

In this thesis, we have proposed a strong energy-aware power management scheme to achieve a target lifetime for OS-based mobile systems. The PM scheme manages energy as the first-class resource all around the system, it consists of two parts: an epoch mechanism that guarantees a user-desired lifetime by partitioning it into a number of epochs and limiting the total energy available in each epoch; and an energy-based fair queuing scheduling algorithm that provides proportional energy using among tasks and supports time-constrain meeting. This PM scheme allows the energy to be managed ahead of other system resources, thus, facilitates the achieving of more stringent and advanced energy-related goals.

Our work in this thesis is focused on the energy-based fair queuing scheduling. Besides of providing proportional power sharing among tasks, an energy-based fair queuing scheduling algorithm should also support time-sensitive tasks in a general purpose operating system. We have extended the starting-time fair queuing (SFQ) to the energy management domain, and proposed the starting-energy fair queuing (SEFQ) based on our energy model. Simulation results show that under SEFQ, tasks can consume energy proportionally to their assigned power shares, and the power share of any time-sensitive task can be protected to be no lower than its desired share in dynamic energy competitions. Also, to avoid missing most of the time constraints and maintain a stable long-term power share, a time-sensitive task should be assigned a power share that is no lower than its maximum long-term power share. In SEFQ, the minimum power share required by a real-time task for meeting its worst-case energy load is sensitive to the energy allocation error. Since the allocation error is a function of the number of active tasks and is difficult to be combined into the computation of the worst-case power share, a conservative power share that may be overly larger than the worst-case power share has to be reserved to a hard real-time task. Moreover, the instability of the worst-case power share requires a timely adjustment on the assigned power share to avoid missing deadlines. The borrowed starting-energy fair queuing (BSEFQ) is proposed to better support time-sensitive tasks by combining a real-time friendly mechanism into the SEFQ. BSEFQ breaks the fairness between time-sensitive tasks and regular tasks by giving dispatching priorities to the time-sensitive tasks; however, the maximum time one time-sensitive task can run with priority is limited. Simulation results show that under BSEFQ the time constraints of real-time tasks can be strictly met without being affected by the energy allocation error, while the long-term power shares of tasks keeps the same as those under SEFQ. BSEFQ flexibly and

effectively supports different types of time-sensitive tasks in a way that the real-time performance of different types of time-sensitive tasks can be traded off depending on their given priorities, but with their minimum performance guaranteed. Moreover, BSEFQ can trade off between the power control and time-constraint meeting by properly setting the maximum time that time-sensitive tasks can run with priority. Therefore, on one side, the energy consumption of time-sensitive tasks of high energy load can be restricted to avoid draining the battery in a high rate and to protect the energy usage of other tasks; on the other side, time constraints can be stringently met when energy is consumed in a normal rate.

In conclusion, we have presented two energy-based fair queuing algorithms, SEFQ and BSEFQ, and showed that they can manage energy in a way that tasks can proportionally consume energy according to their assigned power shares. Besides, with parameters properly set, the two algorithms, especially BSEFQ, can well support time-sensitive tasks in general operating systems without bringing the risk to starve the regular tasks. SEFQ and BSEFQ are promising algorithms that can be combined with the epoch mechanism to manage energy as the first-class resource and guarantee a user-specified battery lifetime to a target application in OS-based mobile systems.

6.2. Future research

In the future, we will improve our work from three aspects: the algorithm design, the simulation test-bench, and the implementation on Linux.

The epoch-combined energy-based fair queuing scheduling can be improved by resolving several issues we currently have. First, more efforts have to be made to resolve the updating problem of system virtual energy caused by the warping mechanism in BSEFQ. Although we have proposed several solutions in section 3.6.2 and implemented one of them in the SystemC test bench, how well it works is unknown because the situation that a new time-sensitive task joins the competition is not considered in our simulations. Based on the SystemC test-bench, different approaches will be assessed to find the best solution to the updating problem of the system virtual energy. Second, as have been pointed out at the end of section 5.3, to dynamically extend the CPU active time to the whole epoch, the possibility of combining DVFS and application adaptation into the epoch-combined SEFQ will be explored. Note that without combining DVFS, application adaptation or any other energy-efficient algorithm, the current epoch-combined energy-based fair queuing scheduling does not save any joule of energy itself, but provides a framework to manage energy as a first-class

resource and facilitates the achievement of more advanced energy goals. Third, it is also possible that the energy allocated to one epoch is not exhausted at the end of the epoch. Then, the remaining energy has to be reclaimed, but how the reclaimed energy should be properly reallocated to the rest epochs should be explored to minimize the residual energy in the battery and thus, maximize the application performance within the target lifetime goal.

The current test-bench has some limitations as have already been pointed out in section 4.4. The improved test-bench should support variable-length service quanta and successive simulation over different epochs; also meaningful energy and power values should be employed to produce more convincing results. In a word, the test-bench will be refined to support simulations on the latest scheduling algorithms.

The proposed scheduling algorithm will be implemented in the Linux kernel and tested in a battery-powered system running on a commercial board named BeagleBoard. Many works have to be done to achieve that. First, since in a real-system the energy is consumed in different components, the energy model has to be extended to manage energy in a system-wide manner by taking into account the energy consumption on other modules, such as the memory, network and data bus; the scheduling algorithms, however, do not require many modifications due to the fact that energy is managed as the first-class resource. Second, energy measurement and accounting tools are required to measure the energy consumption on devices and map them to specific tasks. An instrumented BeagleBoard is currently under its final functional test and soon it will be available for use. The instrumented BeadgleBoard can measure the energy consumption in different modules of the hardware while the system is running, further work is required to find or design a proper tool to achieve energy accounting. Also, to enable the application of our scheduling algorithms on non-instrumented commercial boards, research works on performance monitor counter-based energy estimation are concurrently carried out in our research group. Finally, the code of Linux kernel has to be modified to add our scheduling algorithms.

7 References

[1] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Ecosystem: Managing Energy as a First Class Operating System Resource. Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOSX), ACM Press, pp. 123–132, 2002.

[2] R. Neugebauer and D. McAuley. Energy is Just another Resource: Energy Accounting and Energy Pricing in the Nemesis OS. In Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII), May 2001.

[3] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Currentcy: A Unifying Abstraction for Expressing Energy. Proc. USENIX Ann. Technical Conf., USENIX, pp. 43–56, 2003.

[4] J. Lorch and A. Smith. Software Strategies for Portable Computer Energy Management," IEEE Personal Commun., vol. 5, pp. 60–73, June 1998.

[5] L. Benini and G. De Micheli. Dynamic Power Management: Design Techniques and CAD Tools. Norwell, MA: Kluwer, 1998.

[6] L. Benini, A. Bogliolo, S. Cavallucci, and B. Riccó. Monitoring System Activity for OS-directed Dynamic Power Management. In Proceedings of the International Symposium on Low Power Electronics and Design, pp. 185–190, Aug. 1998.

[7] T. Pering, T. D. Burd, and R. W. Brodersen. The Simulation and Evaluation of Dynamic Scaling Algorithms. In Proceedings of the International Symposium on Low Power Electronics and Design, August 1998.

[8] W. Yuan and K. Nahrstedt. Energy-efficient Soft Real-time CPU Scheduling for Mobile Multimedia Systems. Proc. 19th ACM Symp. Operating Systems Principles (SOSP 03), ACM Press, pp. 149–163, 2003.

[9] J. Lorch and A. Smith. Improving Dynamic Voltage Scaling Algorithms with PACE. In Proc. of ACM SIGMETRICS 2001 Conference, June 2001.

[10] P. Pillai and K. G. Shin. Real-time Dynamic Voltage Scaling for Low-power Embedded Operating Systems. In Proc. of 18th Symposium on Operating Systems Principles, Oct. 2001.

[11] J. Flinn and M. Satyanarayanan. Energy-aware Adaptation for Mobile Applications. In Symposium on Operating Systems Principles (SOSP), pages 48–63, December 1999.

[12] H. Zeng, C. Ellis, and A. R. Lebeck. Experiences in Managing Energy with ECOSystem, IEEE Pervasive Computing, January-March 2005.

[13] C. S. Ellis. The Case for Higher-Level Power Management. In Proceedings of the 7th Workshop on Hot Topics in Operating Systems, Rio Rico, AZ, March 1999.

[14] A. Vahdat, C. Ellis, and A. Lebeck. Every Joule is Precious: The Case for Revisiting Operating System Design for Energy Efficiency. In Proceedings of the 9th ACM SIGOPS European Workshop, September 2000.

[15] B. Noble, M. Price, and M. Satyanarayanan. A Programming Interface for Application-aware Adaptation in Mobile Computing. Computing Systems, 8(4):345-363, 1995.

[16] J. Flinn and M. Satyanarayanan. PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications. Second Workshop on Mobile Computing Systems & Applications (WMCSA'99), New Orleans, February, 1999.

[17] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile Application-aware Adaptation for Mobility. In Proceedings of the 16th ACM Symposium on Operating Systems and Principles, pages 276–287, Saint-Malo, France, October 1997.

[18] C. A. Waldspurger and W. E. Weihl. Stride Scheduling: Deterministic Proportional Share Resource Management. Technical Memorandum, MIT/LCS/TM-528, Laboratory for CS, MIT, July 1995.

[19] J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In Proceedings of the 11th IEEE Real-time Systems Symposium, pp. 166-171, December, 1989.

[20] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. Journal of the ACM (JACM), Volume 20 Issue 1, Pages 46 - 61, Jan. 1973.

[21] M. Dertouzos. Control Robotics: The Procedural Control of Physical Processors. In Proceedings of the IFIP Congress. Stockholm, Sweden, pp. 807-813, 1974.

[22] V. Baiceanu, C. Cowan, D. McNamee, C. Pu, and J. Walpol. Multimedia Applications Require Adaptive CPU Scheduling. In Proceedings of the IEEE RTSS Workshop on Resource Allocation Problems in Multimedia Systems. Washington, DC, 1996.

[23] M. Dertouzos. Control Robotics: The Procedural Control of Physical Processors. In Proceedings of the IFIP Congress. Stockholm, Sweden, pp. 807-813, 1974.

[24] J. Nieh and M.S. Lam. The Design, Implementation and Evaluation of SMART: A scheduler for Multimedia Applications. In Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles. ACM, St. Malo, France, pp. 184-197, 1997.

[25] C.W. Mercer, S. Savage, and H.Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In Proceedings of the IEEE International Conference on Multimedia Computing and Systems, Boston, MA, PP. 90-99, 1994.

[26] G. Coulson, A. Campbell, P. Robin, G. Blair, M. Papathomas, and D. Hutchinson. The Design of a QoS Controlled ATM Based Communications System in Chorus. IEEE Journal of Selected Areas in Communications (JSAC) 13, 4 (May), pp. 686-699, 1995.

[27] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. IEEE Journal of Selected Areas in Communications (JSAC) 14, 7 (Sept.), pp. 1280-1297, 1996.

[28] M. B. Jones, D. Rosu, and M. C. Rosu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles. St. Malo, France, pp. 198-211, 1997.

[29] D. K. Y. Yau, S. S. Lam. Adaptive Rate-controlled Scheduling for Multimedia Applications, IEEE/ACM Transactions on Networking (TON), v.5 n.4, pp. 475-488, Aug. 1997

[30] J. Nieh and M. S. Lam. A SMART Scheduler for Multimedia Applications. ACM Transactions on Computer Systems, 21(2), pp. 117-163, May 2003.

[31] I. Stoica, H. Abdel-wahab, and K. Je
ay. On the Duality between Resource Reservation and Proportional Share Resource Allocation, In Proc. of Multimedia Computing and Networking, pp. 207-214, 1997.

[32] A. K. Parekh and R. G. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case. IEEE/ACM Transactions on Networking 1, 3 (June), 344-357, 1993.

[33] S.J. Golestani. A Self-Clocked Fair Queueing Scheme for High Speed Applications. In Proceedings of INFOCOM'94, 1994.

[34] P. Goyal, H. M. Vin, and H. Cheng. Start-Time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. IEEE/ACM Transaction on Networking, Vol. 5, No. 5, October 1997.

[35] D. Stiliadis and A. Varma. Efficient Fair Queuing Algorithms for Packet-Switched Networks. IEEE/ACM Transactions on Networking, Vol 6, No 2, pp. 175-185, April, 1998.

[36] I. Stoica, H. Abdel-Wahab, K. Jeffay, and S. K. Baruah. A Proportional Share Resource Allocation Algorithm for Real-time, Time-shared Systems. In Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96), pages 288-299, Dec. 1996.

[37] P. Goyal, X. Guo, and H. M. Vin. 1996. A hierarchical CPU scheduler for multimedia operating systems. Proc. Second USENIX Symposium on Operating System Design and Implementation (OSDI), pp 107-122, 1996.

[38] K. J. Duda and D. R. Cheriton. 1999. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general purpose scheduler. In Proceedings of the 17th ACM Symposium on Operating System Principles, Dec. 1999.

[39] D. Stiliadis and A. Varma, Latency-rate Servers: A General Model for Analysis of Traffic Scheduling Algorithms, In Proc. IEEE INFOCOM ’96, San Francisco, CA, pp. 111–119, Apr. 1996.

[40] L. Zhang. VirtualClock: A New Traffic Control Algorithm for Packet Switching Networks, Proceeding of the ACM SIGCOMM’90. pp. 19-29. September 1990.

[41] J .C. R. Bennet and H. Zhang. Hierarchical Packet Fair Queuing Algorithms, IEEE/ACM Transactions on Networking, Vol. 5, No 5, pp. 675-689. October 1997.

[42] A. K. J. Parekh. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks. Ph. Thesis, Laboratoty for Information and Decision Systems, Massachusetts Institute of Technology, February 1992.

[43] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. Proc. Sigcomm '89, 19(4):1-12, September 1989.

[44] J. C. R. Bennet, J.C.R., and H. Zhang. WF2Q: Worst-Case Fair Weighted Queuing, Proceedings of IEEE Infocom’96, San Francisco, pp. 120-128, March 1996

[45] D. Stilliadis and A. Verma. Frame-based fair queueing: A new traffic scheduling algorithm for packet-switched network. Technical Report USCS-CRL-95-39, University of California at Santa Cruz, July 1995.

[46] S. Suri, G. Varghese and G. Chandranmenon. Leap Forward Virtual Clock: A New Fair Queuing Scheme with Guaranteed Delays and Throughput Fairness. Proceedings of IEEE Infocom’97, Kobe, Japan, pp. 557-565, 1997.

[47] F.Mkhiussi and A.Francini. Minimum-Delay Self-Clocked Fair Queuing Algorithm for Packet-Switched Networks”, IEEE INFOCOMM, vo13, pp.1112-1121, 1998

[48] C. Wang, K. Long, X. Gong, S. Cheng. Effective Fairness Queuing Algorithms. Proc. of ICON 2000, pp. 294 - 301, 2000.

[49] C. Wang, K. Long, X. Gong, S. Cheng. SWFQ: A Simple Weighted Fair Queueing Scheduling Algorithm Forhighspeed Packet Switched Network. Proc. of ICC 2001, vol. 8, pp. 2343 - 2347, 2001.

[50] B.H. Choi et al., Rate Proportional SCFQ (RP-SCFQ) Algorithm for High-Speed Packet-Switched Networks. ETRI Journal., vol. 22, no. 3, pp. 1-9, Sept. 2000.

[51] H. Halabian, H. Saidi and R. Changiz. LVT-SCFQ: A Modified Self Clocked Fair Queueing Algorithm for Broadband Networks. Proceedings of the 3rd International Conference on Broadband Communications, Information Technology & Biomedical Applications BroadCom 2008. 23- 26 November. Pretoria, Gauteng, South Africa. pp. 175-180, 2008.

[52] D. Y. Kwak, N. S. Ko, B. Kim and H. S. Park. A New Starting Potential Fair Queuing Algorithm with O(1) Virtual Time. ETRI Journal, vol.25, no.6, Dec. 2003, pp.475-488.

[53] H. Halabian, H. Saidi, FPFQ: A Low Complexity Fair Queueing Algorithm for Broadband Networks, Proc. of ICTTA 2008, pp. 1-6, April 2008.

[54] J. Davin and A. Heybey. A simulation study of fair queueing and policy enforcement, Comput. Commun. Rev., vol. 20, pp. 23–29, Oct. 1990.

[55] P. Goyal, S. S. Lam, and H. M. Vin. Determining End-to-End Delay Bounds In Heterogeneous Networks. In ACM/Springer-Verlag Multimedia Systems Journal (to appear), 1996. Also appeared in the Proceedings of the Workshop on Network and Operating System Support for Digital Audio and Video, April 1995

[56] Stiliadis D., and A. Varma. Rate-Proportional Servers: A Design Methodology for Fair Queueing Algorithms. IEEE/ACM Transactions on Networking, Vol. 6, No 2, pp. 164-174. April 1998.

[57] D. Stiliadis and A. Varma, A general methodology for designing efficient traffic scheduling and shaping algorithms, in Proc. IEEE INFOCOM'97, San Francisco, CA, Apr. 1997, pp. 326–335.

[58] L. Abeni, G. Lipari, and G. Buttazzo. Constant bandwidth vs. proportional share resource allocation. In Proceedings of the 1999 IEEE International Conference on Multimedia Computing and Systems (ICMCS '99), June 1999.

[59] J. S. Goddard and J. Tang. EEVDF Proportional Share Resource Allocation Revisited, in Work-in-Progress Sessions of the 21st IEEE Real-Time Systems Symposium (RTSSWIP00), Nov. 2000.

[60] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. Proceedings of the First Symposium on Operating Systems Design and Implementation, November 1994.

[61] R. E. Al-Ouran, Linux Implementation of a New Model for Handling Task Dynamics in Proportional Share Based Scheduling Systems, Ohio University, 2010. http://etd.ohiolink.edu/send-pdf.cgi/AlOuran%20Rami%20R.pdf?ohiou1275681466

[62] J. Regehr. Some Guidelines for Proportional Share CPU Scheduling in General-purpose Operating Systems, In The 22nd IEEE Real-Time Systems Symposium (RTSS 2001), London, UK, December 3-6 2001.

[63] A. Bavier, L. Peterson, and D. Mosberger. BERT: A Scheduler for Best-Effort and Realtime Paths. Technical Report TR-602-99, Department of Computer Science, Princeton University, 1999.

[64] J. Nieh, C. Vaill, and H. Zhong. Virtual-Time Round-Robin: An O(1) Proportional Share Scheduler. In Proceedings of the General Track: 2002 USENIX Annual Technical Conference, (Berkeley, CA, USA), pp. 245-259, USENIX Association, 2001.

[65] A. Greenberg and N. Madras. How Fair is Fair Queuing. The Journal of ACM, 39(3):568-598, July 1992.

[66] I. Stoica, H. Abdel-Wahab. Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation. Technical Report TR-95-22, CS Dpt., Old Dominion Univ., Nov. 1995.

[67] K. Lee. Performance Bounds in Communication Net-works With Variable-Rate Links. In Proceedings of ACM SIGCOMM'95, pages 126-136, 1995.

[68] F. Bellosa. The Benefits of Event-driven Energy Accounting in Power-sensitive Systems. In Proceedings of the 9th workshop on ACM SIGOP European workshop.(Kolding, Denmark, Sep 17-20 2000). ACM New York, NY, USA, pp. 37-42, September, 2000. DOI= http://doi.acm.org/10.1145/566726.566736

[69] INTEL. Mobile Power Guidelines 2000 Rev 1.0, Dec 1998

[70] T. L. Martin. Balancing Batteries, Power and Performance: System Issues in CPU Speed-Setting for Mobile Computing. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1999.